MORAL

ISSN 0265-2919
80p 41

THE HOME COMPUTER ADVANCED COURSE

MAKING THE MOST OF YOUR MICRO

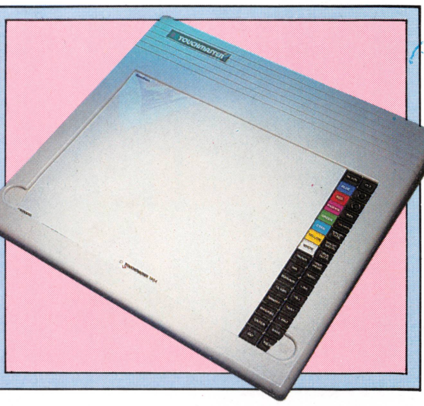
An ORBIS Publication
IR £1 Aus $1.95 NZ $2.25 SA R1.95 Sing $4.50 USA & Can $1.95

CONTENTS

[illegible]

Next Week

- We begin a survey of robots currently available on the market, starting with those priced under £200.
- The 6809 machine code series comes to an end as we publish the final part of the debugger project.
- The final touch: we examine the Touchmaster graphics tablet, which can be interfaced to a wide range of home micros.



QUIZ

1) Apart from the keyboard, what other new feature has been added to the Spectrum+, and where is it?
2) What kind of motor are we using to build the robot and why is this type more suitable for the purpose?
3) What kind of addressing would we expect to use to access an array in the computer's memory?
4) What is the difference between an equation processor and a spreadsheet?

Answers To Last Week's Quiz

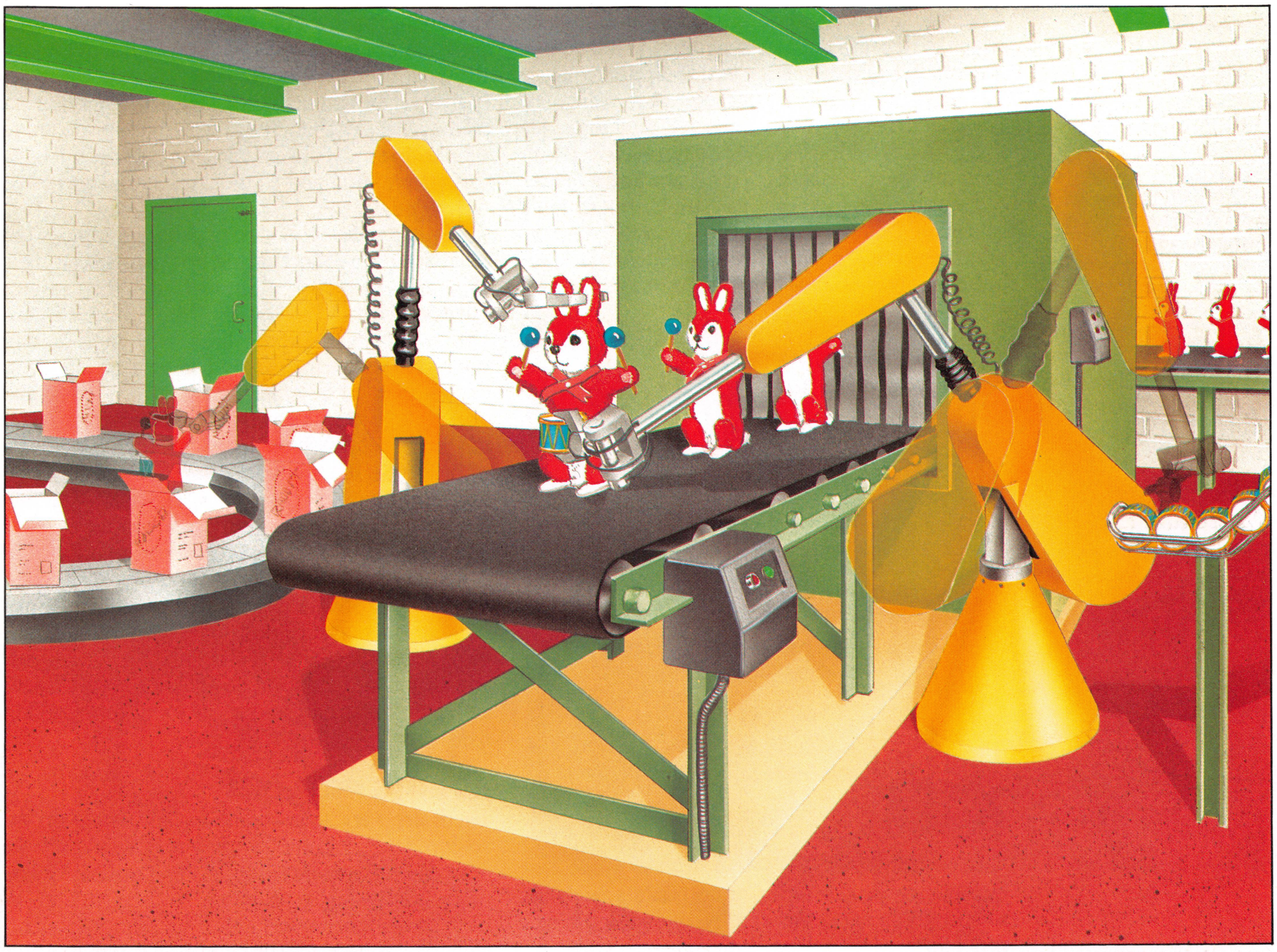
1) If the software written for one model of computer in a manufacturer's range will work on all succeeding models in the range, it is said to be 'upward compatible'.
2) 'Daisy chaining' means sharing a data bus or power supply among several devices by connecting the output of one device to the input of the next, and so on around the chain.
3) In Lotus 1-2-3 commonly-used sequences of keystrokes can be saved and named as a sub-program; this is called a 'keyboard macro'.
4) INVENTORY is a command used in most adventure games to display the list of your possessions.

Editor Mike Wesley; **Technical Editor** Brian Morris; **Production Editor** Catherine Cardwell; **Art Editor** Claudia Zeff; **Chief Sub Editor** Robert Pickering; **Designer** Julian Dorr; **Art Assistant** Liz Dixon; **Staff Writer** Stephen Malone; **Sub Editor** Steve Mann; **Consultant Editor** Steve Colwill; **Contributors** Geoff Bains, Harvey Mellar, Mike Curtis, Steve Colwill, Chris Naylor, Tony Harrington, Steve Malone, Steve Mann; **Software Consultants** Pilot Software City; **Group Art Director** Perry Neville; **Managing Director** Stephen England; **Published by Orbis Publishing Ltd: Editorial Director** Brian Innes; **Project Development** Peter Brookesmith; **Executive Editor** Maurice Geller; **Production Controller** Peter Taylor-Medhurst; **Designed and produced by Bunch Partworks Ltd; Editorial Office** 14 Rathbone Place, London W1P 1DE; **© APSIF Copenhagen 1984; © Orbis Publishing Ltd 1984: Typeset by Universe; Reproduction by Mullis Morgan Ltd; Printed in Great Britain by Artisan Press Ltd, Leicester**

HOW TO OBTAIN ISSUES AND BINDERS FOR THE HOME COMPUTER ADVANCED COURSE – Issues can be obtained by placing an order with your newsagent or direct from our subscription department. If you have any difficulty obtaining any back issues from your newsagent, please write to us stating the issue(s) required and enclosing a cheque for the cover price of the issue(s). **AUSTRALIA** – please write to: Gordon & Gotch (Aus) Ltd, 114 William Street, PO Box 767G, Melbourne, Victoria 3001. **MALTA, NEW ZEALAND & SOUTH AFRICA** – Back numbers are available at cover price from your newsagent. In case of difficulty, write to the address given for binders.
UK/EIRE – Price: 80p/IR£1. Subscription: 6 months: £23.92. 1 Year: £47.84. Binder: please send £3.95 per binder, or take advantage of our special offer in early issues. **EUROPE** – Price: 80p. Subscription: 6 months air: £37.96. Surface: £31.46. 1 year air: £75.92. Surface: £62.92. Binder: £5.00. Airmail: £5.00. **MALTA** – Obtain binders from your newsagent or Miller (Malta) Ltd, MA Vassalli Street, Valetta, Malta. Price: £3.95. **MIDDLE EAST** – Price: 80p. Subscription: 6 months air: £43.94. Surface: £31.46. 1 year air: £87.88. Surface: £62.92. Binder: £5.00. Airmail: £8.31. **AMERICAS/ASIA/AFRICA** – Price: US/CAN$1.95/80p. Subscription: 6 months air: £51.74. Surface: £31.46. 1 year air: £103.48. Surface: £62.92. Binder: £5.00. Airmail: £9.44. **SOUTH AFRICA** – Price: SA R1.95. Obtain binders from any branch of Central News Agency or Intermag, PO Box 57394, Springfield 2137. **SINGAPORE** – Price: Sing $4.50. Obtain binders from MPH Distributors, 601 Sims Drive, 03-07-21, Singapore 1438. **AUSTRALASIA/FAR EAST** – Price: 80p. Subscription: 6 months air: £55.38. Surface: £31.46. 1 year air £110.76. Surface: £62.92. Binder: £5.00. Airmail: £9.84. **AUSTRALIA** – Price: Aus$1.95. Obtain binders from First Post Pty Ltd, 23 Chandos Street, St Leonards, NSW 2065. **NEW ZEALAND** – Price: NZ$2.25. Obtain binders from your newsagent or Gordon & Gotch (NZ) Ltd, PO Box 1595, Wellington.
ADDRESS FOR BINDERS AND BACK ISSUES – Orbis Publishing Limited, Orbis House, Bedfordbury, London WC2 4BT. Telephone 01-379 6711. Cheques/postal orders should be made payable to Orbis Publishing Limited. Binder prices include postage and packing and prices are in stirling. Back issues are sold at the cover price, and we do not charge carriage in the UK.
NOTE – Binders and back issues are obtainable subject to availability of stocks. Whilst every attempt is made to keep the price of the issues and binders constant, the publishers reserve the right to increase the stated prices at any time when circumstances dictate. Binders depicted in this publication are those produced for the UK and Australian markets only. Binders and Issues may be subject to import duty and/or local taxes, which are not included in the above prices unless stated.
ADDRESS FOR SUBSCRIPTIONS – Orbis Publishing Limited, Hurst Farm, Baydon Road, Lambourn Woodlands, Newbury Berks, RG16 7TW. Telephone: 0488-72666. All cheques/postal orders should be made payable to Orbis Publishing Limited, Postage and packaging is included in subscription rates, and prices are given in sterling.

COVER PHOTOGRAPHY BY IAN McKINNELL

PERFECT COPY



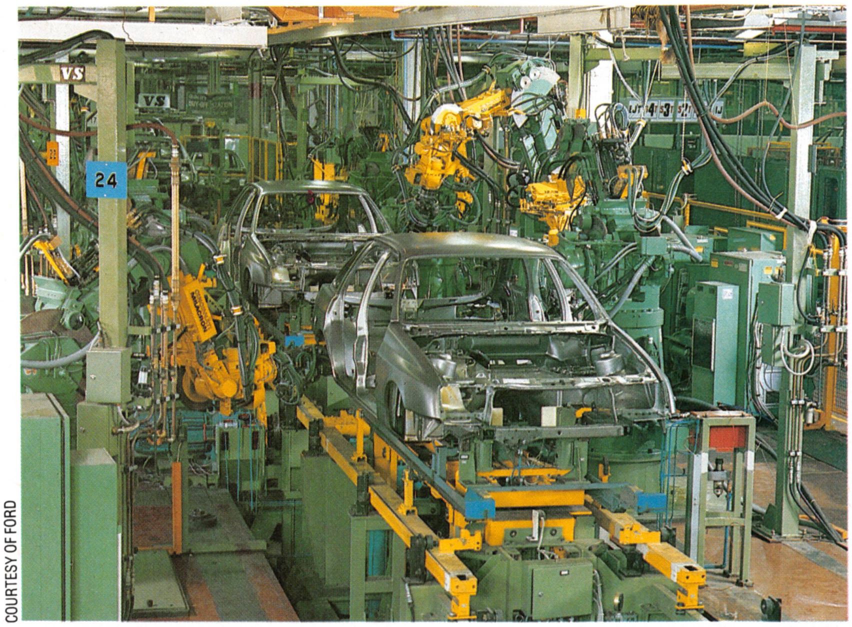
We've spent some time in this series considering the different methods that may be used to allow robots to act 'intelligently'. However, the typical computer user is unlikely to have access to a robot and therefore cannot put these ideas into action. The simple answer is to simulate a robot's behaviour by using a computer.

The development of computer technology has led to an increasing use of simulations: computer 'models' may be constructed that will faithfully mimic events in the 'real' world. Most people are familiar with the idea of flight simulators — extremely complex devices that enable trainees to gain flying experience without having to pilot a real aircraft. But many other activities can benefit from computer simulation — business forecasting, engineering operations and physical processes of all types can very easily be simulated on a computer model. In some cases, the computer model can carry out experiments that would be too dangerous to attempt by any other means. It might be vitally important to discover what happens in a nuclear power station when coolant leaks from the reactor. In this instance, it would obviously be impossible to use a real power station for the experiment, so a computer simulation is used. If the model is sufficiently detailed, it is then possible to see exactly what would happen if the leak occurred.

Similarly, in robotics, computer simulations are used to design new robots. It is obviously possible to proceed by trial and error — building a robot, watching how it behaves, and then making any necessary modifications — but this is time-consuming and expensive. A computer simulation allows you to design your robot and monitor its actions without spending money and without the physical labour involved in making frequent design changes.

Moving Together
When robot arms are engaged in a common task there is a real need for choreography in order that they do not interfere with one another. Here, one arm must pick up and hold the toy in position while the other picks up and fixes a drum; the first arm then places the completed toy in its packing box. If the movement of the arms is properly managed then conveyor belts can be placed in any arrangement that suits the rest of the assembly, whereas a human operator's ergonomic requirements would limit this freedom

Take as an example a car assembly line, on which a team of robots works on the cars as they pass by. All you want to do is to program the robots so that they will assemble the cars in the correct manner. However, programming the robots takes time, and money is lost each time the production line is halted. You might decide to set up a dummy assembly line with some brand new robots with which to develop the new programs. But this, too, is expensive and can easily lead to another snag — the problem of robot choreography, which we considered earlier in this series. It is vital to ensure that robots working together do not interfere with each other's movements. This is not just a matter of convenience — a large industrial robot capable of moving heavy loads could easily damage another robot if it should run into it. And it's not only the robots that may be damaged — an improperly programmed robot that spends its time welding car doors shut would soon prove a problem!


Chorus Line
The complexity of movement and synchronisation required in real robotics applications can be clearly seen in this section of the Ford Sierra assembly line

COURTESY OF FORD

The obvious answer is to carry out computer simulations of each robot's actions so that the user can see how they will interact. This way the cost is low and nothing is damaged. Once the simulation is complete and everything looks satisfactory, the programs that have been developed can easily be transferred to the real robots, which can then be safely left to carry out their designated tasks.

In this article, we will demonstrate the principle of robot simulation with a program, Robot Arm, that simulates a 'pick and place' robot arm with two degrees of freedom. It has no sensors, so you must guide it yourself, controlling the shoulder and elbow joints and the grab mechanism in the end effector, in order to pick up an object and then place it somewhere else. Additionally, you should refer back to the maze-solving program detailed on page 722, which demonstrates how a robot may be progammed to find its way to the centre of a maze. This program is, in effect, a computer simulation of how a 'real' robot would attempt to reach its goal. It mimics the actions of a robot fitted with a simple touch sensor, and it finds the correct path by simply advancing into empty spaces until it meets a dead end, whereupon it returns to the last junction it encountered and then tries a new route. This is hardly a sophisticated model, but it does show how a computer program can be used to simulate a robot's movements. The 'robot' in the program obeys a fixed set of rules and 'maps out' the environment. If, within the program, the robot had direct and immediate access to the positions of all the maze components, it would be able to move directly to its goal. In our program, it does not have this information and so must use a trial and error technique.

Similarly, the Robot Arm program mimics the behaviour of a robot that has no sensors at all. This program contains a model of the robot's environment and a model of the arm itself, and you must ensure that these two models will interact only as they would in real life. So you cannot pick up an object with the arm unless the arm is positioned correctly. And you cannot move the arm below floor level, as that would be impossible if the robot arm was real. Although we are using computer graphics, in which one line (representing the arm) may easily cross another line (the floor), for an accurate simulation it is essential that these lines do not cross. And when the robot drops an object, that object must not remain in its current position — your simulation must allow gravitational effects. If this is ignored, you would certainly not be able to develop a safe simulation of a pick and place robot for handling eggs!

ADDING REALISM

There are very few limitations on what can be achieved using computer simulation — and, in most cases, the more complex the simulation is the more fascinating it becomes. Such a simulation can be even more entertaining than simply playing with 'real' robots, for the simple reason that, using a simulation, you can design any robot you like; programming the correct details of the robot and its world can lead to a better understanding of robots and of the way in which the physical world works. Look again at the Robot Arm program. You will see that when the robot drops an object it falls to the ground and stays there. To make the model even more realistic, the program could be altered so that the object accelerates as it falls, thus obeying the law of gravity. And perhaps, on hitting the ground, it should bounce? The possibilities are many, and the program is there for you to adapt, adding new and more realistic features to make your simulation as lifelike as possible.

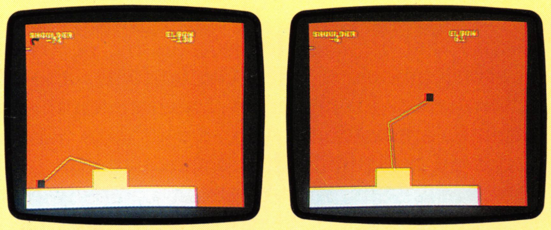
Designing computer simulations can be very similar to developing computer games software. The big difference is that a simulation must represent the real world as accurately as possible. Achieving this accuracy may be difficult but, once achieved, the simulation can be considerably more satisfying than merely playing a computer game.

Pick And Place

This program simulates a robot arm that is able to reach about, pick objects up and place them down in another location. Your task is simply to pick up the ball and then drop it. The arm is designed to use co-ordinates of revolution with two degrees of freedom: a shoulder joint and an elbow joint. The shoulder joint can rotate through 180° and the elbow joint through 360°.

The program is controlled using the following keys: S indicates that you want a shoulder movement; E indicates that you want an elbow movement; the K and H keys indicate whether you wish the joint of the arm to be rotated clockwise or anticlockwise. Each press rotates the shoulder joint through 6°, or the elbow through 12°. U indicates that you want the arm to attempt to pick up the ball. This will only be successful if you have managed to manipulate the arm within reach of the ball. F indicates that you want the robot to drop the ball

```
   4 REM ******SPECTRUM**********
   5 REM * ROBOT ARM SIMULATION*
   6 REM ******SPECTRUM**********
  10 CLS : PRINT "ROBOT ARM": PRINT : PRINT "THE
 CONTROLS FOR THE ROBOT ARM                 ARE:"
  15 PRINT "S- SELECT SHOULDER ROTATION": PRINT
"E- SELECT ELBOW ROTATION": PRINT "K-ROTATE JOIN
T CLOCKWISE"
  20 PRINT "H-ROTATE JOINT ANTICLOCKWISE": PRINT
 "U-GRAB BALL": PRINT "F-DROP BALL"
  25 PRINT AT 20,11; FLASH 1;"hit a key": PAUSE
0: RANDOMIZE
1000 GO SUB 9500:   REM init
1100 GO SUB 5000:   REM input
1200 GO SUB 6000:   REM crash
1400 STOP
1500 REM *update jointxy *************
1550 LET ex=l1*SIN hd1: LET ey=l1*COS hd1
1560 LET hx=sx+ex: LET hy=sy+ey
1650 LET wx=l2*SIN hd2: LET wy=l2*COS hd2
1660 LET hx=hx+wx: LET hy=hy+wy
1690 RETURN
2000 REM *   draw arm   ************
2020 INK acol
2050 PLOT  INVERSE rubout;sx,sy
2100 DRAW  INVERSE rubout;ex,ey: DRAW  INVERSE r
ubout;wx,wy: IF blup THEN  LET br=FN r(hy): LET
bc=FN c(hx): GO SUB 2500
2490 RETURN
2500 REM * draw ball    **************
2600 INK bcol
2650 PRINT AT br,bc;"█ "(rubout+1)
2740 RETURN
2750 REM * drop ball  ****************
2800 LET rubout=1: GO SUB 2000: LET rubout=0
2820 LET k=INT (xh*RND): IF k>=xs THEN  IF k<=xs
+wd THEN  GO TO 2820
2850 LET br=FN r(y0+4): LET bc=FN c(k): LET blup
=0: GO SUB 2000: GO SUB 2500
2990 RETURN
3000 REM *   rotate    ***************
3100 LET rubout=1: GO SUB 2000
3120 LET t1=dirn*sr*a1: LET t2=t1+dirn*er*a2
3150 LET hd1=hd1+t1: LET hd2=hd2+t2
3200 GO SUB 1500
3300 IF ABS hd1>p2 THEN  LET ok=0
3320 LET pt=POINT (hx,hy)
3340 IF pt<>0 THEN  LET ok=0: IF br=FN r(hy) AND
 bc=FN c(hx) THEN  LET ok=2
3400 LET rubout=0: GO SUB 2000
3450 INK bacol: PRINT AT 21,0;s$;AT 21,2;FN d(hd
1);AT 21,26;FN d(hd2)
3490 RETURN
5000 REM *   input     *****************
5100 IF INKEY$<>"" THEN  GO TO 5100
5120 FOR l=0 TO 1 STEP 0
5150 LET a$=INKEY$: IF a$>="A" AND a$<="Z" THEN
 LET a$=CHR$ (CODE a$+32)
5200 IF a$="s" THEN  LET sr=1: LET er=0
5220 IF a$="e" THEN  LET er=1: LET sr=0
5250 IF a$="k" THEN  LET dirn=1: GO SUB 3000
5270 IF a$="h" THEN  LET dirn=-1: GO SUB 3000
5300 IF a$="u" THEN  IF ok=2 THEN  LET blup=1
5320 IF a$="f" THEN  IF blup THEN  GO SUB 2750
5400 IF NOT ok THEN  LET l=2
5450 NEXT l
5490 RETURN
6000 REM *   crash    *****************
6100 PRINT AT 8,12; FLASH 1;"!!crash!!": BEEP .5
,-5: BEEP 1,-16: RETURN
9000 REM *   draw base   ***********
9050 PAPER pacol: CLS
9100 INK grcol
9120 FOR k=0 TO y0: PLOT 0,k: DRAW xh,0: NEXT k
9200 INK bacol: LET xs=(xh-wd)/2
9220 FOR k=y0+1 TO y0+ht
9240 PLOT xs,k: DRAW wd,0
9260 NEXT k
9300 INK bcol: GO SUB 2500: INK acol
9400 PRINT AT 20,1;"SHOULDER";AT 20,26;"ELBOW"
9490 RETURN
9500 REM *  init   ******************
9550 DEF FN d(x)=INT (x*180/PI)
9560 DEF FN r(x)=21-INT (x/8)
9570 DEF FN c(x)=INT (x/8)
9600 DIM s$(32): LET xl=0: LET yl=0: LET xh=254:
 LET yh=174
9620 LET y0=23: LET wd=60: LET ht=23
9630 LET blup=0: LET bc=2: LET br=FN r(y0+4)
9640 LET grcol=3: LET bacol=2: LET acol=1: LET b
col=6: LET pacol=7
9650 LET sx=xh/2: LET sy=y0+ht+2: LET l1=(yh-ht-
y0-2)/2: IF l1>xh/4 THEN  LET l1=xh/4
9660 LET l2=l1: LET hx=0: LET hy=0
9670 LET p2=PI/2: LET a1=PI/32: LET a2=2*a1
9680 LET hd1=0: LET hd2=p2
9690 LET sr=1: LET er=0: LET dirn=1: LET rubout=
0: LET ok=1
9750 GO SUB 9000: GO SUB 1500: GO SUB 2000
9790 RETURN
```



IAN McKINNELL

Straight From The Shoulder
Our robot arm simulation program allows you to move a two-joint arm in two dimensions and pick up an object with it. When the object is dropped the program places it randomly on the floor. The display shows the vertical angles made by the upper and lower arms

On the BBC Micro make the following additions and changes:

```
   4 REM************BBC***************
   5 REM*   ROBOT ARM SIMULATION     *
   6 REM************BBC***************
   7 MODE 1:COLOUR 130:COLOUR 1:CLS
  25 PRINTTAB(15,20)"HIT A KEY":A$=GET$
1000 GOSUB 9600
2020 GCOL 0,acol
2050 MOVE sx,sy
2100 PLOT rubout,ex,ey:PLOT rubout,wx,wy:IF
blu p THEN br=hy:bc=hx:GOSUB 2500
2190 RETURN
2200 REM********GRAB THE BALL*********
2250 blup=1:rubout=3:GOSUB 2500
2300 rubout=1:GOSUB 2000
2600 GCOL 0,bcol:MOVE bc,br
2650 PLOT 0,0,bsz:PLOT 80+rubout,bsz,0
2700 PLOT 0,0,-bsz:PLOT 80+rubout,-bsz,0
2800 rubout=3:GOSUB 2000:rubout=1
2820 k=INT(xh*RND(1)):IF k>=xs THEN IF k<=xs+wd
THEN GOTO 2820
2850 br=y0+5:bc=k:blup=0:GOSUB 2000:GOSUB 2500
3100 rubout=3:GOSUB 2000
3340 IF pt<>pacol-128 THEN ok=0:IF pt=bcol THEN
ok=2
3400 rubout=1:GOSUB 2000
3450 COLOUR bacol:PRINTTAB(0,3)s$;TAB(4,3);FNd
( hd1);TAB(27,3);FNd(hd2)
5100 IF INKEY$(0)<>"" THEN GOTO 5100
5150 a$=INKEY$(0):IF a$>="A"ANDa$<="Z"THENa$=CH
R$(ASC(a$)+32)
5300 IF a$="u" AND ok=2 THEN GOSUB 2200
5400 IF ok=0 THEN l=2
6100 PRINTTAB(12,3)"!!CRASH!!":SOUND 1,-15,48,1
0:SOUND 1,-15,4,20:RETURN
9050 GCOL 0,pacol:COLOUR pacol:CLS
9100 GCOL 0,grcol
9120 FOR k=0 TO y0:MOVE 0,k:DRAW xh,k:NEXT k
9200 GCOL 0,bacol:xs=(xh-wd)/2
9240 MOVE xs,k:DRAW xs+wd,k
9300 MOVE bc,br:GOSUB 2500:COLOUR acol
9400 PRINTTAB(1,2)"SHOULDER";TAB(26,2)"ELBOW"
9600 s$="":x1=0:y1=0:xh=1000:yh=1000
9620 y0=100:wd=200:ht=100
9630 blup=0:bsz=wd/5:bc=40:br=y0+5
9640 grcol=3:bacol=2:acol=2:bcol=0:pacol=129
9650 sx=xh/2:sy=y0+ht+2:l1=(yh-ht-y0-2)/2:IF l1
>xh/4 THEN l1=xh/4
9690 sr=1:er=0:dirn=1:rubout=1:ok=1
```

PROBLEM SOLVER

Our spreadsheet series continues with a close look at TK!Solver, a modelling program from the creators of VisiCalc that takes the spreadsheet concept into a new direction: equation processing.

As we have seen in this series, microcomputer spreadsheet programs can be very useful for a variety of mathematical tasks. For the person accustomed to working on large row and column worksheets with a pencil and a calculator, the electronic spreadsheet is an invaluable time and energy saver. Nevertheless, spreadsheets do have significant limitations. The row and column format that is ideal for accounting or other financial models is often cumbersome, and at times useless, for higher level mathematical and scientific applications. And spreadsheets have a very rigid structure for handling equations.

Software Arts, the American company that created VisiCalc, has developed a program called TK!Solver that goes beyond spreadsheets in both form and function. 'TK!' stands for ToolKit, while 'Solver' is the section of code that actually processes equations. Besides differing from spreadsheets in screeen format, TK! offers the following unique features:

Backsolving — Spreadsheet formulae can solve for a single variable only. TK! can solve for any variable in an equation, if it is given enough data to do so;

Iteration — If a value required to solve an equation is missing or unknown, you can input a guess that TK! uses as a starting point. It then solves the equation through a series of successive approximations;

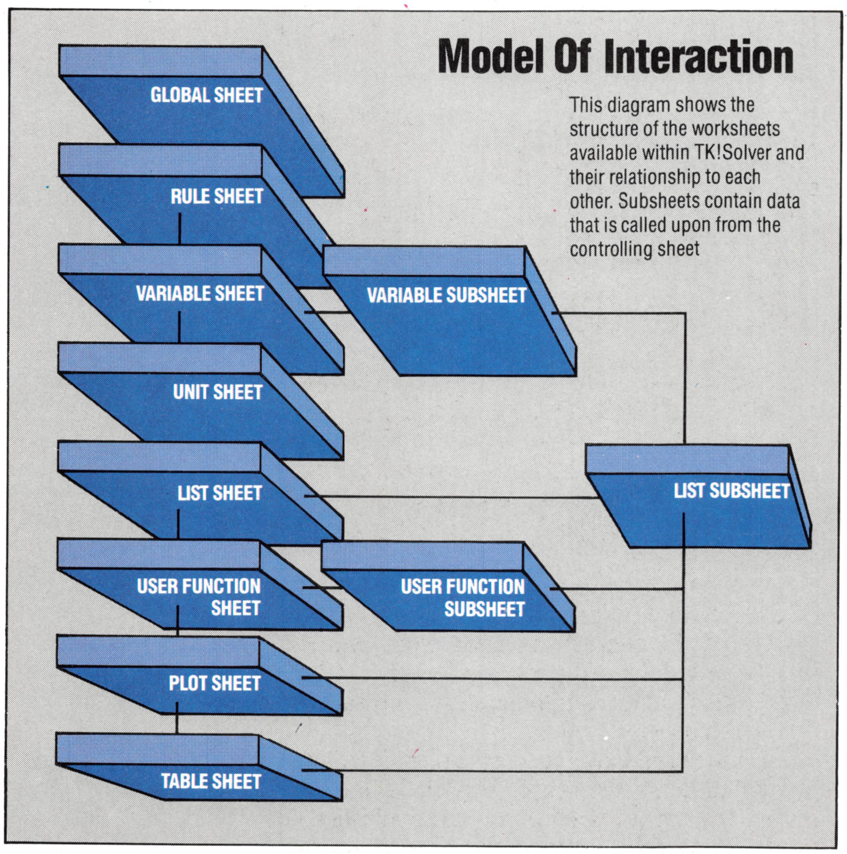
Unit conversion — TK! can convert values from feet to metres, dollars to pounds, etc., instantly from conversion tables;

Mathematical functions — TK! has a large number of these built in.

TK!SOLVER WORKSHEETS

The TK!Solver program operates through three linked worksheets, each with a specific function. The Variable sheet contains the names of all defined variables; columns for the user's input values and the program's output values; a place to indicate associated units, and space for the user to annotate each variable with a comment. The Variable sheet appears at the top of the program's initial display screen. Each variable is also described in detail on a separate variable subsheet. The Rule sheet is used to enter the equations TK! is expected to solve. An equation can be up to 200 characters long, and must conform to standard mathematical conventions of notation and operations. The Rule sheet fills the bottom portion of TK!'s opening screen. The Unit sheet stores the information needed to convert the units of measurement attached to the variables in a model.

TK!Solver uses these three sheets to perform most of its operations. Other sheets include a Global sheet, in which the user can customise some of TK!'s operating procedures; a List sheet that stores an array of values for variables; the User Function sheet, for user-defined functions; and sheets for plotting and printing points or tables of values.



Model Of Interaction

This diagram shows the structure of the worksheets available within TK!Solver and their relationship to each other. Subsheets contain data that is called upon from the controlling sheet

CREATING A MODEL

We will begin by creating a very simple model adapted from the TK!Solver owner's manual that calculates mileage and average speed for an automobile journey, and converts the values from imperial to metric units. In TK!'s opening display, we find the cursor in the Rule sheet at the bottom of the screen. We begin by defining the variables in appropriate equations, so we type:

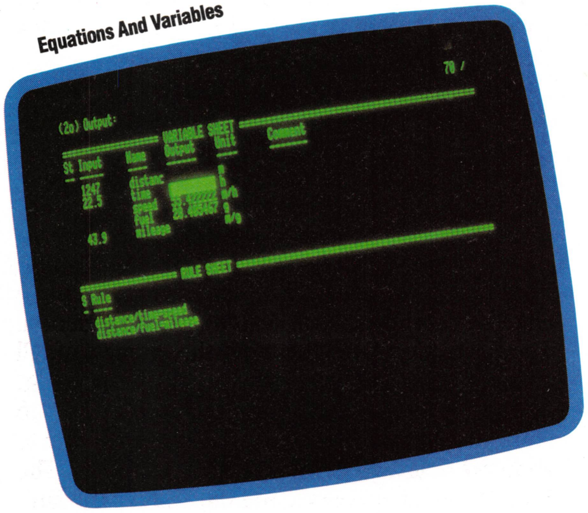
```
distance/time=speed
```

and press Return. Initially, TK! is set to read variable names from equations directly onto the Variable sheet above. TK! evaluates the equation and prints the variables in the Name column on the Variable sheet in the same order as they appear in the equation. Then an asterisk is displayed in the Status column next to the equation. The asterisk means that the equation is unsatisfied, because no values have been input on the variable sheet. We then enter the second equation in the same way:

```
distance/fuel=mileage
```

When this equation has been entered, all five of the variables defined will be listed in the Name column on the Variable sheet as shown below.

Equations And Variables



```
                                                   78 /
(2o) Output:
                  VARIABLE SHEET
St Input    Name     Output     Unit      Comment
   1247     distanc  [illegible] m
   22.5     time     [illegible] h
            speed    28.405467  m/h
            fuel                g
   43.9     mileage             m/g

                   RULE SHEET
S Rule
  distance/time=speed
  distance/fuel=mileage
```

Press the semi-colon (;) key to move the cursor from the Rule sheet to the Variable sheet, into which we can now enter values. The cursor appears in the input column next to our first variable, distance. The following values are entered by typing them in the appropriate space, then pressing Return or the down arrow key.

INPUT	**NAME**	**OUTPUT**
500	distance	
8.5	time	
	speed	
14	fuel	
	mileage	

The speed and mileage values are left blank for TK! to solve. Their calculated values will be displayed in the OUTPUT column. To solve for speed and mileage, press the exclamation mark (!), which TK! calls the 'action' key. TK! will display the phrase Direct Solver above the Variable sheet because the program has been given all the data required to find a direct solution. Shortly, the values for speed and mileage will be displayed as output. We can delete the values previously input and obtain a figure for distance by giving TK! new values for speed and time, or for mileage and fuel.

UNIT CONVERSION

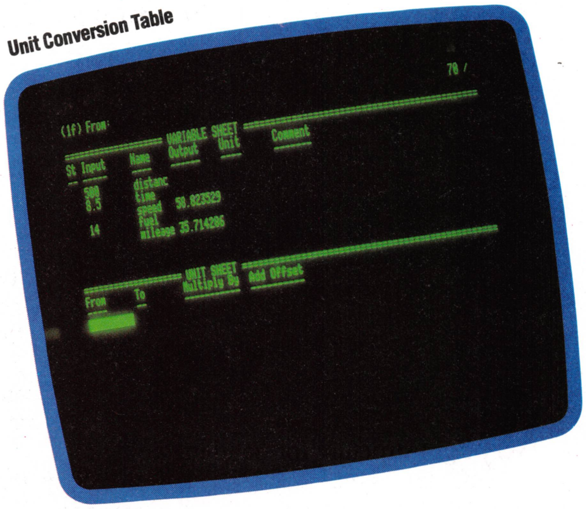
The values entered in our model so far have no units attached. We cannot simply type miles, or gallons, in the unit column on the Variable sheet, because units may not be used until they have been defined. We move the cursor into the Rule sheet by pressing the semi-colon (;) key and then typing =U. TK! replaces the Rule sheet in the bottom window with the Unit sheet.

The Unit sheet has four columns:

```
From      To      Multiply by      Add Offset
```

The cursor is displayed below the word From. We can then enter the units we want TK! to know and the conversion values as shown below.

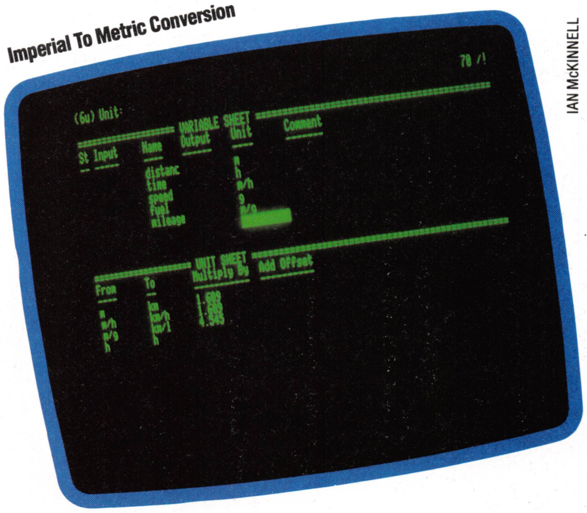
Unit Conversion Table



```
                                                   78 /
(1f) From:
                  VARIABLE SHEET
St Input    Name     Output     Unit      Comment
   500      distanc
   8.5      time
            speed    58.823529
            fuel
   14       mileage  35.714286

                   UNIT SHEET
From      To      Multiply By   Add Offset
```

Press =R to restore the Rule sheet, then ; to move into the Variable sheet. We can now enter the defined names into the Unit column — m for distance; h for time; m/h for speed; g for fuel, and m/g for mileage. Blank all the current values and replace them with these: 1,247 for distance; 22.5 for time; and 43.9 for mileage. Press ! to solve, and the metric values are displayed.

Imperial To Metric Conversion



```
                                                   78 /!
(6u) Unit:
                  VARIABLE SHEET
St Input    Name     Output     Unit      Comment
            distanc             m
            time                h
            speed               m/h
            fuel                g
            mileage             m/g

                   UNIT SHEET
From      To      Multiply By   Add Offset
m         km      1.609
m/h       km/h    1.609
m/g       km/l    4.545
h         h
```

IAN McKINNELL

Now place the cursor over m in the unit column and type km for kilometres. Press Return, and TK! will automatically convert the value of 1,247 for distance from miles to kilometres, so the value of distance changes to 2006.423.

We have used only a few of the facilities offered by TK!Solver in this simple model. In the next instalment, we will look at a more sophisticated model, using TK!'s function and plotting abilities.

DRESSED TO QL

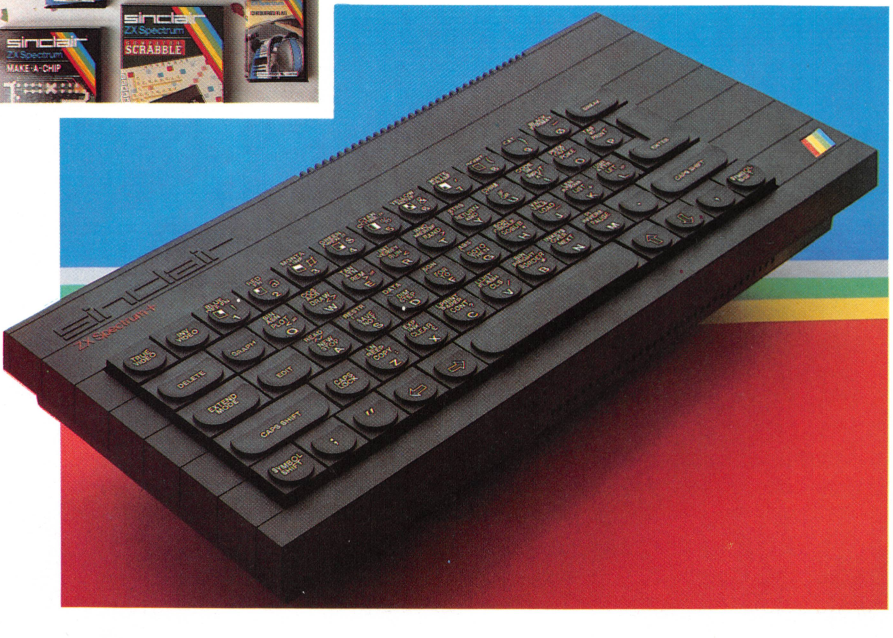
The Sinclair Spectrum has proved to be the most successful home computer ever in the UK. However, the machine has begun to appear a little outdated and unimaginative next to stylish rivals like the Amstrad CPC 464 and Commodore Plus/4, and Sinclair has responded by dressing the old machine up in new clothes.

At its launch in the spring of 1982, the Sinclair Spectrum offered outstanding value for money. Its only real rivals were the Vic-20, with a meagre 3.5 Kbytes of user memory, and the Texas TI99 4A, which sold at twice the price. At £175, the Spectrum was an instant hit with first-time buyers as well as being the natural choice of the thousands of micro enthusiasts who had outgrown their ZX80s and ZX81s. The new machine had an astounding 48 Kbytes of memory, used a good BASIC, and offered eight colours for graphics, as well as a primitive sound facility. The keyboard, too, was a vast improvement on the 'touch-sensitive' flat plastic sheet of the ZX81. Initially available by mail order only, the Spectrum was an instant success, and rapidly became the country's best-selling micro.

In the two and a half years since the Spectrum's launch, Sinclair's competitors produced a range of machines to challenge the Spectrum's market dominance. Despite having a decidedly inferior BASIC, the Commodore 64 was the most successful challenger; it offered more memory (although machine code was needed to make the most of this), superb sound and a 'real' keyboard with typewriter-style moving keys. The BBC Micro, too, offered superior specifications, but its £400 price prevented it from being a serious threat, and its manufacturers chose not to lower its price. However, a succession of Commodore price cuts reduced the cost of the 64 from an initial £340 to £150; Sinclair reacted by dropping the Spectrum to £130.

By this time, the Spectrum keyboard — once such an attraction — was now a decided drawback. The machine's software base was unsurpassed, and many 'serious' packages were produced for it. However, trying to use word processing programs with the Spectrum keyboard was like typing with mittens on. Many users, therefore, invested in 'proper' keyboards, and this trend accelerated when the long-awaited Interface 1/Microdrive unit finally appeared. It soon became apparent that the micro users of 1984 were no longer prepared to accept the Sinclair idea of what constituted an acceptable input device.

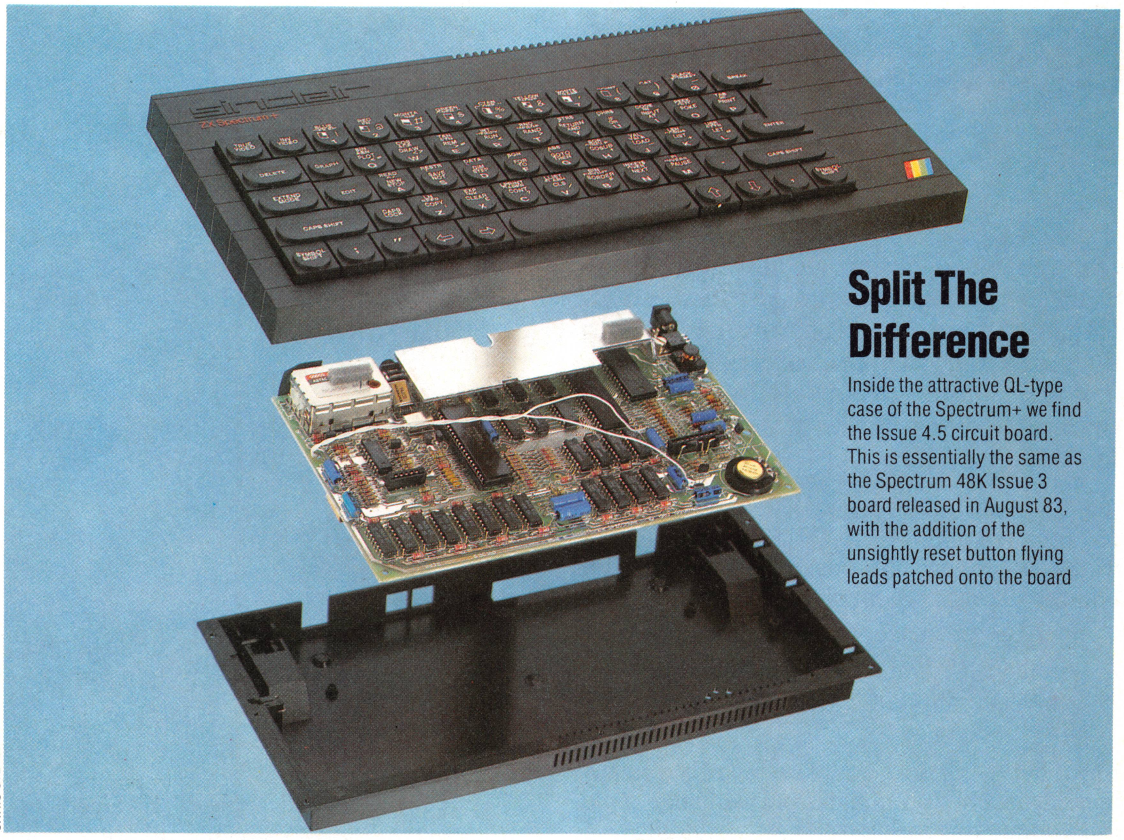
Six Of One
Following the current fashion for 'bundling' software with home micros, Sinclair includes an impressive software six-pack with the Spectrum+ (and the Spectrum 48K). It comprises a word processor, spreadsheet, two games, and two graphics packages — at least £30 worth of good-quality software



SPECTRUM FACELIFT

Sinclair Research's response has been to give the Spectrum a facelift. The Spectrum+ is essentially the same machine, but housed in a cut-down QL keyboard, with a few extra keys, a Reset switch and a couple of retractable legs. All of the peripherals produced for the older version should work with the 'Plus', but Sinclair has failed to take the opportunity to bring the Spectrum's performance more in line with the competition by improving the sound, or providing a monitor socket or a built-in Interface 1. The sound capabilities of the machine are now its greatest handicap. The two legs do allow a little more volume to escape from the machine's base but this is more an annoyance than a convenience as it means that the LOADing and SAVEing noises are also magnified. The Spectrum's pathetic BEEPing is still woefully inadequate.

The Spectrum+ measures 319 by 149 by 38 mm ($12\frac{1}{2}$ by $5\frac{7}{8}$ by $1\frac{1}{2}$in). The new design makes programming easier by providing extra keys for 'Extended' and graphics mode, true and inverse video, Delete and Break keys, and separate keys for commonly used punctuation symbols like the semi-colon, quotes, comma and full stop. An extra Symbol Shift key has also been added, and the cursor keys are now allocated new places alongside a small Space bar. All the old key combinations still work. For veteran Sinclair users the new design may cause a few problems. In particular, the new Edit key is situated next to the 'A' key; if this is hit by mistake when a long


CHRIS STEVENS

Split The Difference

Inside the attractive QL-type case of the Spectrum+ we find the Issue 4.5 circuit board. This is essentially the same as the Spectrum 48K Issue 3 board released in August 83, with the addition of the unsightly reset button flying leads patched onto the board

SPECTRUM+

PRICE

£180 including six demonstration cassettes

DIMENSIONS

319×149×38 mm

MEMORY AND INTERFACES

Same as Spectrum 48K, BASIC resident, full software compatibility

KEYBOARD

58 sculpted keys (including true space bar); membrane keyboard (see page 503)

DOCUMENTATION

Coloured illustrated tutorial manual with user guide cassette

STRENGTHS

The wealth of existing Spectrum software, user groups and specialist publications are enviable attractions

WEAKNESSES

Despite the new keys and the legs, the keyboard's 'feel' and action are still a major weakness

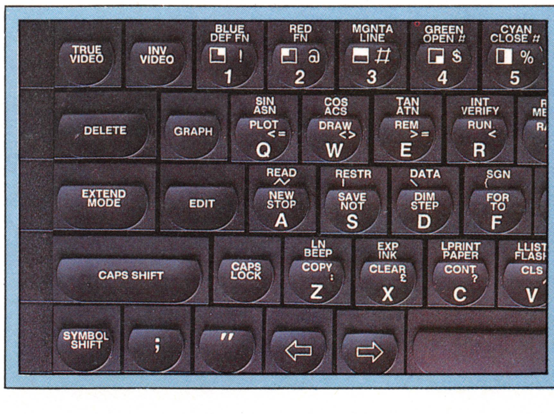
program line is being entered, the line will be lost.

As part of the repackaging, Spectrum+ purchasers receive a 'six-pack' of programs — Psion Chess, Make-a-Chip, Scrabble, Chequered Flag, Vu-3D and the excellent Tasword Two word processing software. All of these programs are of a very high standard. Unfortunately, the same cannot be said for the new Spectrum documentation, which, although beautifully presented, lacks the depth of the old Spectrum manual. The publishers do suggest, however, that as users become more proficient with the machine, they can send away for more comprehensive manuals, costing £7 each.

For games playing, the new version is certainly better than the original, but then most keen players will have invested in a joystick and interface. Both the Kempston and Fuller interfaces work with the Spectrum+, although, as on the older machine, the use of the Fuller Soundbox may stop some software from running. The Kempston Centronics interface performs perfectly, as well, as does the Wafadrive mass storage system.

The Spectrum+ is certainly an improvement on the original Spectrum, but Sinclair's idea of what constitutes a reasonable keyboard is not going to meet with universal approval. Although it might be considered a clever move on Sinclair's part to utilise QL technology in a bid to make the Spectrum more attractive to buyers, with a £50 price increase the new keyboard should be compared with the already available add-ons. Using such a criterion it cannot be considered good value for money. The new machine certainly looks more stylish, but the keys, despite being 'sculpted' to make keying easier, are unresponsive and too crowded.

For first-time buyers, the Spectrum+ is certainly worth considering, but the facilities offered may not be considered worth the extra £50. The cynical might say that Sinclair has introduced the Spectrum+ purely as a way of increasing prices — it would hardly be surprising if the original model was soon phased out. In fact, the introduction of this model is a strangely half-hearted gesture by Sinclair; it would surely have made more sense to have cut the price of the older version (and of the Interface 1/Microdrive package) and left it alone. On the other hand, Sinclair could have increased the price slightly more and included all the things the Spectrum *really* needs, such as proper sound facilities, a moving-key keyboard, monitor socket, and perhaps a built-in Microdrive. But then it would never have been ready in time for Christmas . . .



Key Features

Whether you think the Spectrum+'s QL-like keyboard an improvement or not depends very much on your typing style; the provision of separate keys for the more important functions, however, is a real benefit. The original Spectrum key sequences (e.g. [SYM SHIFT]+[0] for the semi-colon) duplicate the new keys' effects

IAN McKINNELL

ENDGAME

Our project to design an adventure game using LOGO has reached its final stage. Having defined the various locations used in the game, and written procedures to move between them, we conclude by developing routines to deal with the necessary detail of the adventure story.

Our Shrine of Zoltoth game has only two 'perils' incorporated in it. In ROOM.4, the player is faced with a large unfriendly snake, and the program branches to a special 'peril' procedure:

```
TO SNAKE.ATTACKS
   PRINTL [[THERE IS A HUGE SNAKE] [SLOWLY
   MOVING TOWARDS YOU!]]
END
```

The other 'peril' does not place the player in any immediate physical danger, but certainly could cause long-term problems:

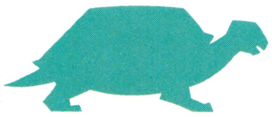
```
TO GATE
   PRINTL [[A GREAT GOLDEN GATE CLOSES
   BEHIND YOU] [CUTTING OFF THE SOUTHERN
   EXIT]]
   MAKE "PERILS []
   MAKE "EXIT.LIST [[N 7] [E 8]]
END
```

Other considerations need to be taken into account at certain places in the program. The GET procedure must be altered so that you cannot pick up the ring if you are carrying the sword.

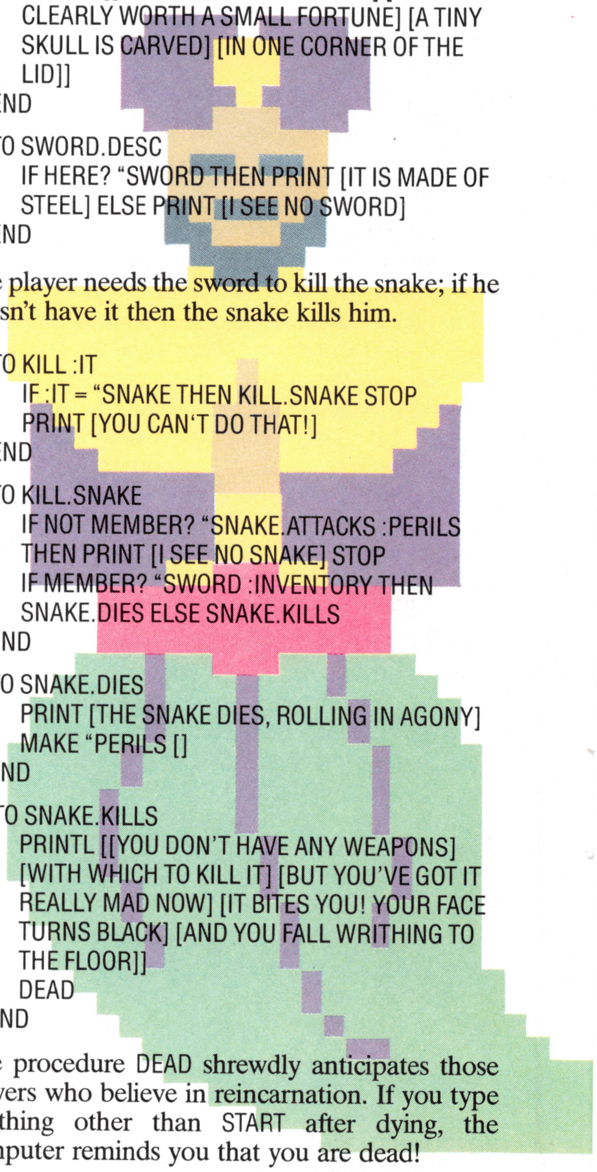
```
TO GETIT :ITEM
   IF :ITEM = "RING THEN GET.RING STOP
   ADD.TO.INV :ITEM
   REMOVE.FROM.ROOM :ITEM
END

TO GET.RING
   IF MEMBER? "SWORD :INVENTORY THEN
   PRINT [YOU ARE UNABLE TO LIFT
   THE RING] STOP
   ADD.TO.INV :ITEM
   REMOVE.FROM.ROOM :ITEM
END
```

This is the only restriction on the player picking up an object. The following routines allow you to examine whatever it is you are holding.

```
TO EXAMINE :OBJ
   IF :OBJ = "RING THEN RING.DESC STOP
   IF :OBJ = "CHEST THEN CHEST.DESC STOP
   IF :OBJ = "SWORD THEN SWORD.DESC STOP
   PRINT [YOU SEE NOTHING SPECIAL]
END
```



```
TO RING.DESC
   IF HERE? "RING THEN PRINTL [[ON THE RING
   IS A FADED INSCRIPTION:] [R—— —E]] ELSE
   PRINT [I SEE NO RING]
END

TO HERE? :OBJ
   IF MEMBER? :OBJ :CONTENTS THEN OUTPUT
   "TRUE IF MEMBER? :OBJ : INVENTORY THEN
   OUTPUT "TRUE
   OUTPUT "FALSE
END

TO CHEST.DESC
   PRINTL [[IT IS BEAUTIFULLY MADE] [AND
   CLEARLY WORTH A SMALL FORTUNE] [A TINY
   SKULL IS CARVED] [IN ONE CORNER OF THE
   LID]]
END

TO SWORD.DESC
   IF HERE? "SWORD THEN PRINT [IT IS MADE OF
   STEEL] ELSE PRINT [I SEE NO SWORD]
END
```



The player needs the sword to kill the snake; if he doesn't have it then the snake kills him.

```
TO KILL :IT
   IF :IT = "SNAKE THEN KILL.SNAKE STOP
   PRINT [YOU CAN'T DO THAT!]
END

TO KILL.SNAKE
   IF NOT MEMBER? "SNAKE.ATTACKS :PERILS
   THEN PRINT [I SEE NO SNAKE] STOP
   IF MEMBER? "SWORD :INVENTORY THEN
   SNAKE.DIES ELSE SNAKE.KILLS
END

TO SNAKE.DIES
   PRINT [THE SNAKE DIES, ROLLING IN AGONY]
   MAKE "PERILS []
END

TO SNAKE.KILLS
   PRINTL [[YOU DON'T HAVE ANY WEAPONS]
   [WITH WHICH TO KILL IT] [BUT YOU'VE GOT IT
   REALLY MAD NOW] [IT BITES YOU! YOUR FACE
   TURNS BLACK] [AND YOU FALL WRITHING TO
   THE FLOOR]]
   DEAD
END
```

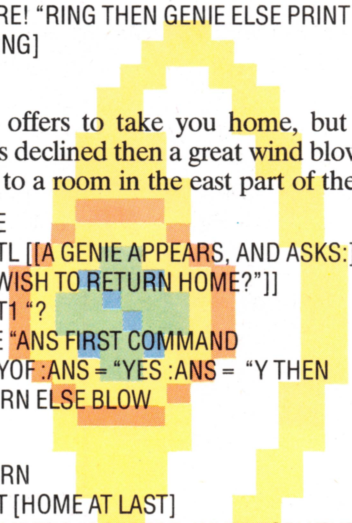
The procedure DEAD shrewdly anticipates those players who believe in reincarnation. If you type anything other than START after dying, the computer reminds you that you are dead!

```
TO DEAD
   PRINT [YOU ARE DEAD!]
```

```
    PRINT1 "?
    MAKE "INPUT COMMAND
    IF ( :INPUT = "START ) THEN START STOP
    PRINT [COME OFF IT!]
    DEAD
END

TO COMMAND
    MAKE "INP REQUEST
    IF :INP = [] THEN PRINT1 "? OUTPUT
    COMMAND
    OUTPUT FIRST :INP
END
```

Rubbing the ring makes the genie appear:

```
TO RUB :OBJ
    IF :OBJ = "RING THEN RUB.RING STOP
    PRINT [IT'S NOW MUCH CLEANER THAN IT
    WAS]
END

TO RUB.RING
    IF HERE! "RING THEN GENIE ELSE PRINT [I SEE
    NO RING]
END
```

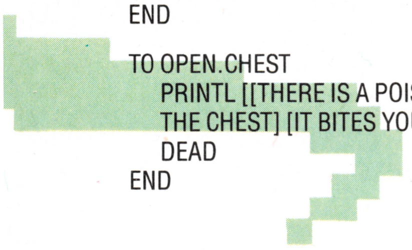


The genie offers to take you home, but if the invitation is declined then a great wind blows you at random to a room in the east part of the cave:

```
TO GENIE
    PRINTL [[A GENIE APPEARS, AND ASKS:] ["DO
    YOU WISH TO RETURN HOME?"]]
    PRINT1 "?
    MAKE "ANS FIRST COMMAND
    IF ANYOF :ANS = "YES :ANS = "Y THEN
    RETURN ELSE BLOW
END

TO RETURN
    PRINT [HOME AT LAST]
    IF MEMBER? "SCEPTRE :INVENTORY THEN
    PRINT [CONGRATULATIONS ON FINDING THE
    SCEPTRE!] ELSE PRINTL [[WELL AT LEAST YOU
    ESCAPED WITH] [YOUR LIFE]]
END

TO BLOW
    PRINT [THERE IS A MIGHTY RUSHING WIND]
    PRINT "
    MOVE1 ( 6 + ( RANDOM 5 ) )
END
```

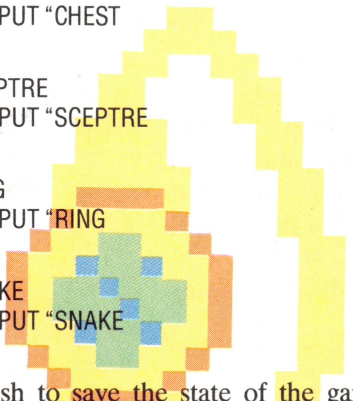
The only thing that can be opened is the chest, and this contains a poisonous spider. The skull on the lid is a warning not to open it — but some people never learn!

```
TO OPEN :OBJ
    IF :OBJ = "CHEST THEN OPEN.CHEST ELSE
    PRINT [YOU CAN'T OPEN IT]
END

TO OPEN.CHEST
    PRINTL [[THERE IS A POISONOUS SPIDER] [IN
    THE CHEST] [IT BITES YOU]]
    DEAD
END
```



Finally, here is a list of all the nouns in the game:

```
TO SWORD
    OUTPUT "SWORD
END

TO CHEST
    OUTPUT "CHEST
END

TO SCEPTRE
    OUTPUT "SCEPTRE
END

TO RING
    OUTPUT "RING
END

TO SNAKE
    OUTPUT "SNAKE
END
```



If you wish to save the state of the game for continuing later, just type SAVE "ADVENTURE, and the entire contents of the workspace are saved. Everything will be restored by READ "ADVENTURE.

In many ways LOGO is an ideal language for programming adventure games. There is one problem, however — there is just not enough room in present day implementations of the language. The game given here barely fits within the memory allocation for the Commodore 64. Any extensions beyond this are going to demand compromises over which words to keep and which to reject.

Logo Flavours

Some versions of MIT LOGO do not have EMPTY?, ITEM, COUNT or MEMBER? Definitions for these were given in the last two instalments (see page 754 and page 775). In all LCSI versions, use:

```
EMPTYP for EMPTY?
LISTP for LIST?
MEMBERP for MEMBER?
TYPE for PRINT1
AND for ALLOF
OR for ANYOF
```

There is a primitive, EQUALP, which tests whether its two inputs are the same. Use this for comparing lists and words in place of the equals sign (which works for lists on some LCSI versions, but not on others).

The IF syntax in LCSI LOGO is demonstrated by:

```
IF EMPTYP :CONTENTS [PRINT [NOTHING
SPECIAL]] [PRINT :CONTENTS]
```

The first list after the condition is performed if the condition is true, and the second if it is false.

On Atari LOGO use SE for SENTENCE, RL for REQUEST, and note that ITEM is not implemented.

The version of the game given in the text was run on the Commodore 64; some other machines may not have enough node space to run all of the game as it stands. If this is the case, then you will have to cut down the size of the game, omitting some of the descriptive words

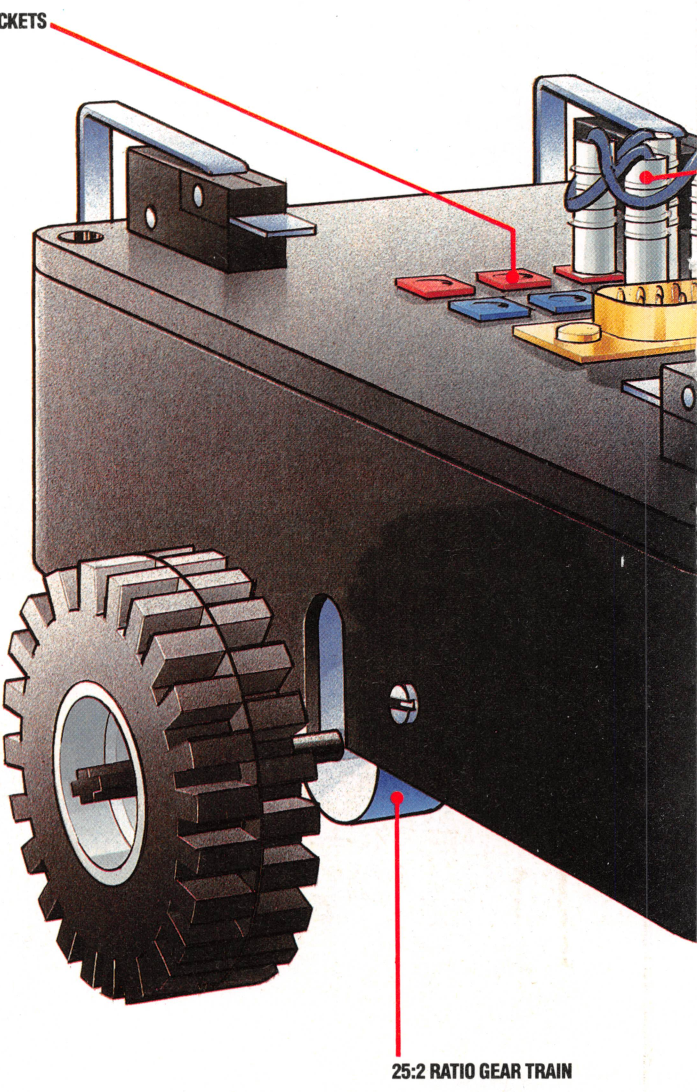
LIZ DIXON

IN THE BEGINNING

In this instalment of Workshop we begin a new project: the construction of an accurately controllable floor robot with proximity and light sensors. In this first section we outline the scheme of the overall project and detail the mechanical construction of the robot body and motor assemblies.

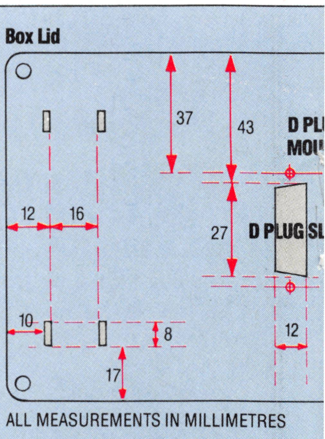
In this new project we shall be constructing and designing software for a floor robot vehicle. The robot will be powered by two stepper motors, driving two wheels through a gearing system. The stepper motors we shall be using can be controlled to turn through discrete steps of 7.5°. Putting the motor drive through a 25:2 ratio gearbox means that the vehicle wheels will be accurately controllable to an axle rotation of 0.6°. As stepper motors operate by turning through a discrete angle each time a pulse is received, they are ideally suited to control by a digital device. We shall be using the computer's user port as our digital control source, allowing us to design simple software to use in conjunction with the robot. In addition to being equipped with stepper motors, the finished robot will have a range of sensors, including proximity sensors and a pair of light sensors to allow the robot to follow a line. As four user port data lines are required to control the vehicle motors, only four more lines are available for inputs from sensors. To allow maximum flexibility, the robot will be fitted with a 'patching' system. This means that different combinations of sensors can be connected to the four available data lines by means of a number of sockets mounted on the robot and the use of short patch leads. For example, one application may require all four proximity sensors, where another might require two proximity sensors and two light sensors. With the patching system the required sensors can be plugged directly into the relevant data input lines.

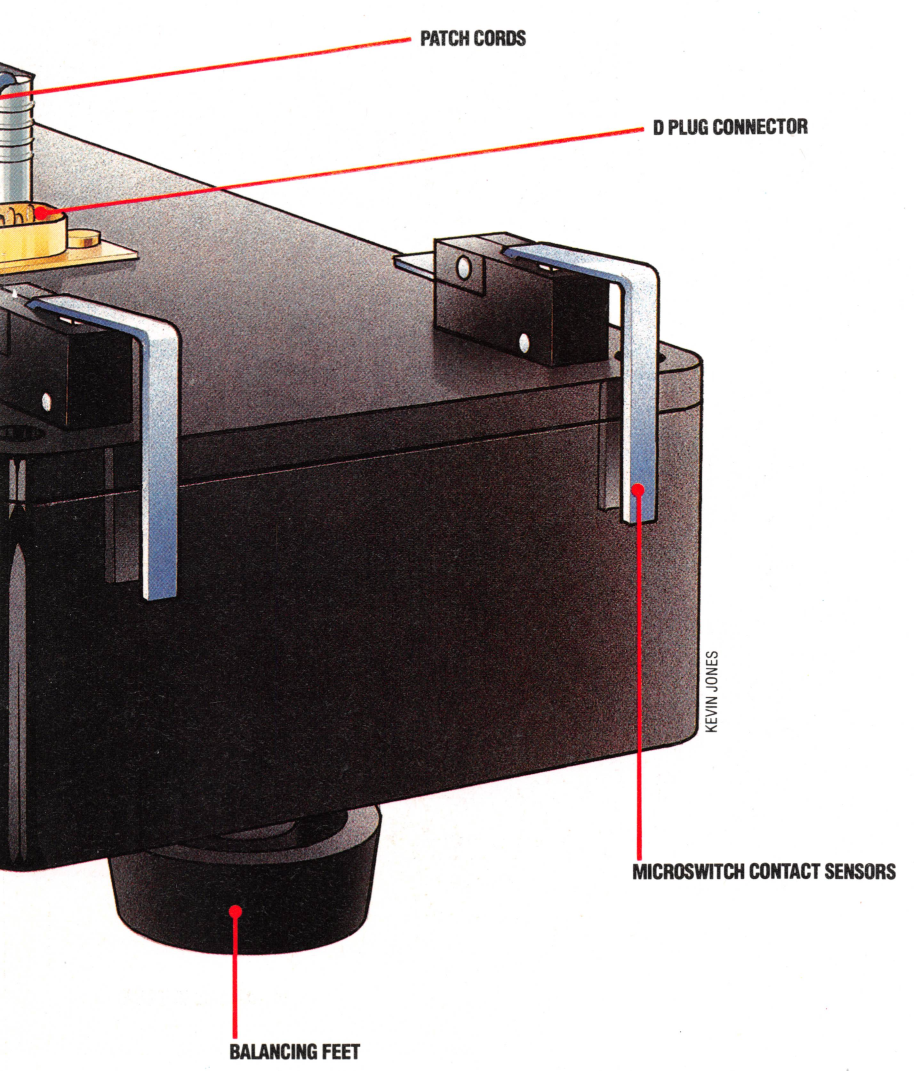
As accurate control of the robot is possible and sensors are fitted, we shall also be undertaking the design of some sophisticated software to allow the creation of an internal map of the robot's immediate environment. We can then start to investigate the intricacies of route-planning and search strategy algorithms. In this first instalment we start the mechanical construction of the robot. This is reasonably straightforward, involving the drilling and cutting of the plastic box that forms the casing and the chassis of the robot; the positioning of the gear train and d-plug mounting holes must be accurate, but the location of the rocker feet is not critical.



Step One

First cut the required holes in the plastic case that is to house and form the chassis of the completed robot. The diagram shows the position and dimensions of the holes needed. Those in the sides and bottom of the box are to take the protruding axles of the drive wheels. The mounting holes for the motor and gearbox block must be in line with one another across the box. The two holes in the bottom of the box are to take the two feet that balance the robot on its two wheels. The hole in the lid is for the D-type socket into which the connecting lead to the computer will plug. To cut the large holes for the gearbox and axles, remove the majority of the plastic with a hot knife or soldering iron. Then bring the hole up to size neatly with a small file

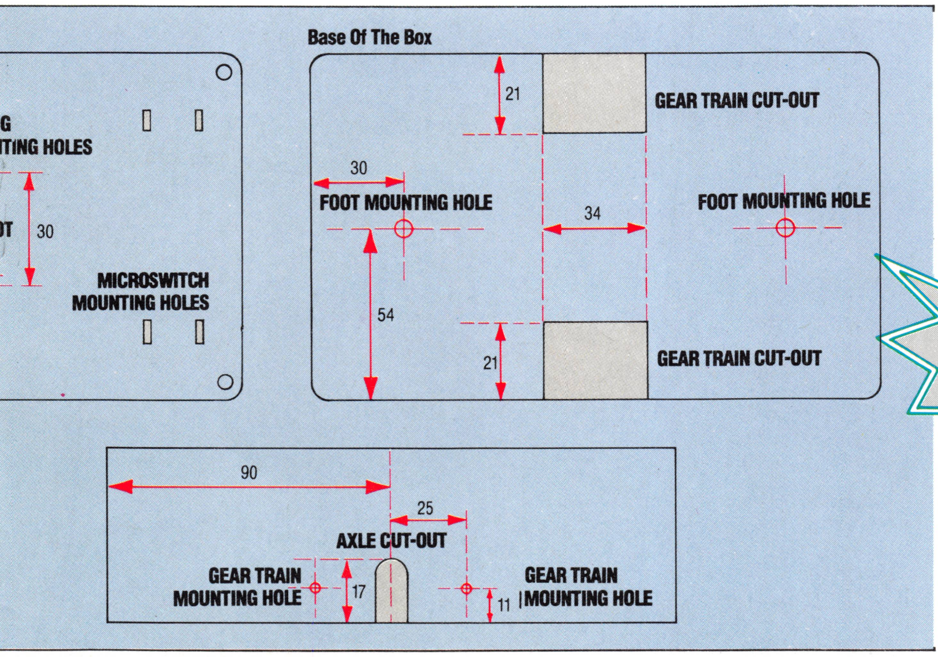




Parts List

No.	Item	Source
RADIO SPARES		
2	SAA 1027 stepper motor drivers	300-237
2	Stepper motors	332-947
2	Synchronous gear boxes 25:2	336-450
MAPLIN		
1	40109	QW67X
3	16-pin DIL sockets	BL19V
2	100 ohm resistor	M100R
2	270 ohm 0.5 watt resistor	S270R
2	0.1 µ F capacitor	YR75S
1	1000 µF 25v capacitors	FB83E
1	24 strip × 50 hole veroboard	FL07H
1	Reel tinned 20 swg wire	BL13P
1	15-way D plug	BK58N
1	15-way D socket	BK59P
1	15-way D cover	BK60Q
1	2.1 mm power socket	RK37S
1	20-way IDC socket	FG87U (BBC)
1	24-way edge connector	BK74R (64)
1	180 × 110 ×55 mm box	LF51F
1	Strip self adhesive pad	HB22V
2	Cabinet feet	FW39N
1	Pack 2BA nuts	BF16S
1	Pack 6BA 0.5in bolts	BF06G
1	Pack 6BA nuts	BF18U
1	Pack M5 25mm bolts	BF32K
1	Pack M5 nuts	BF56L
MISCELLANEOUS		
2	Lego 62mm wheels	Lego 1246
1	Pack technics axles	Lego 1233
4m	12-way ribbon cable	
1m	20-way ribbon cable	(BBC)
1	12-volt 1 amp DC supply	
2	2BA × 4cm bolts	

These parts should cost about £60 in total, which may make the robot more appealing as a group or school project than as an individual effort. The Radio Spares parts can be obtained directly from RS by account holders only; otherwise they can be ordered through other electronics retailers. London readers may like to visit the Robotics Workshop, 121 Ifield Road, SW10 to buy parts and for an interesting insight into the robotics world



WARNING!

The robot consumes a large amount of power; if the power supply has to drive the buffer box as well then the robot is underpowered, and will not move. The robot must, therefore, be connected directly to your computer's user port. If you are not confident of your ability to follow our instructions accurately, you should not attempt this project since mistakes could conceivably cause damage to your computer

Step Two

The motors and gearbox are sold separately and must be assembled. Accompanying the gearbox is a loose small metal gear and a plastic spacer. The gear must be stuck onto the spindle protruding from the motor. Apply some Cyanoacrylate ('Super glue') adhesive to the bore through the gear and place it onto the motor shaft, with the countersink, on the gear, away from the motor. Use the thin end of the spacer to distance the gear correctly from the motor body as shown in the diagram. Leave the glue two or three hours to dry thoroughly

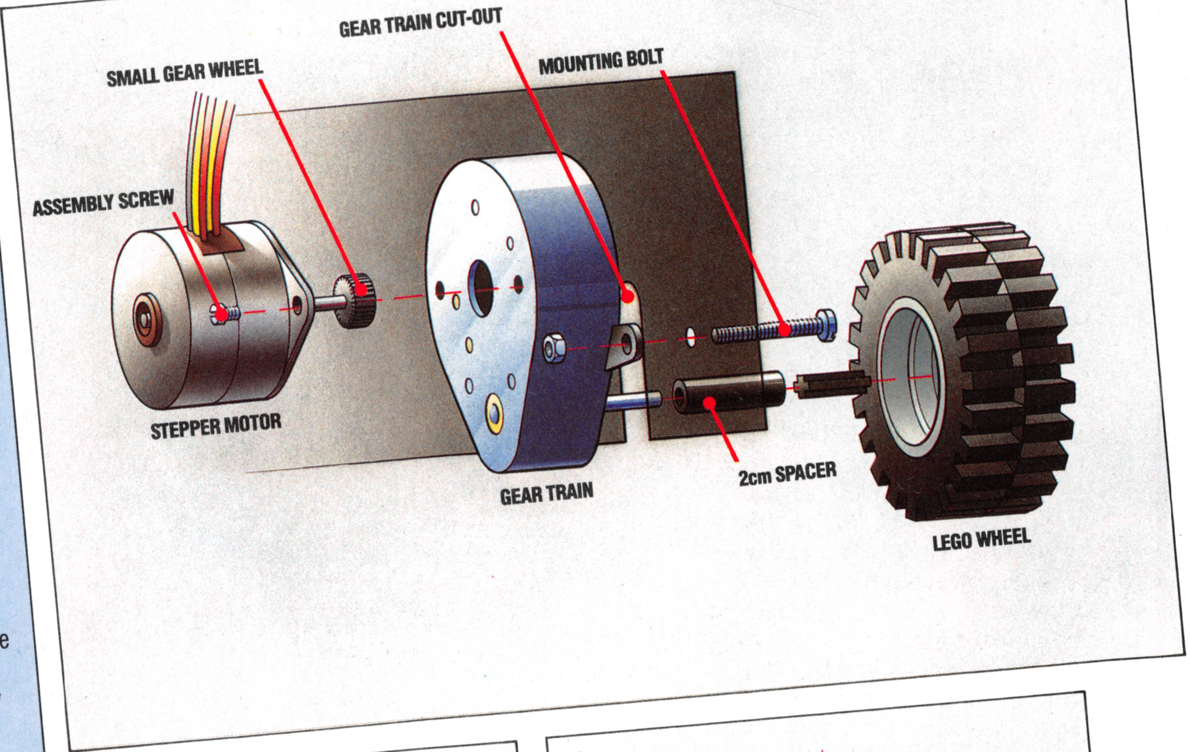
Step Three

Mount the motor (and gear) onto the gearbox with the motor leads towards the wider end of the gearbox. Two screws are provided with the gearbox for this purpose. Be careful to mesh the protruding gear with the internal gears in the gearbox as you press the motor home.

Use the M5 bolts to mount the gearbox in the plastic case. The gearbox is not bolted directly to the case, but held on the bolt which is itself clamped in the case. This makes the gearbox, and therefore the wheel, adjustable.

The wheels mount only onto the special x-sectioned Lego axles. Slip a 2cm length of a plastic ball-point pen case of suitable diameter over the gearbox spindle as a sleeve. Secure it with some Super glue. Now glue a Lego axle into the end of the sleeve. Use the shortest of these axles. The wheels are a 'push' fit onto the axle

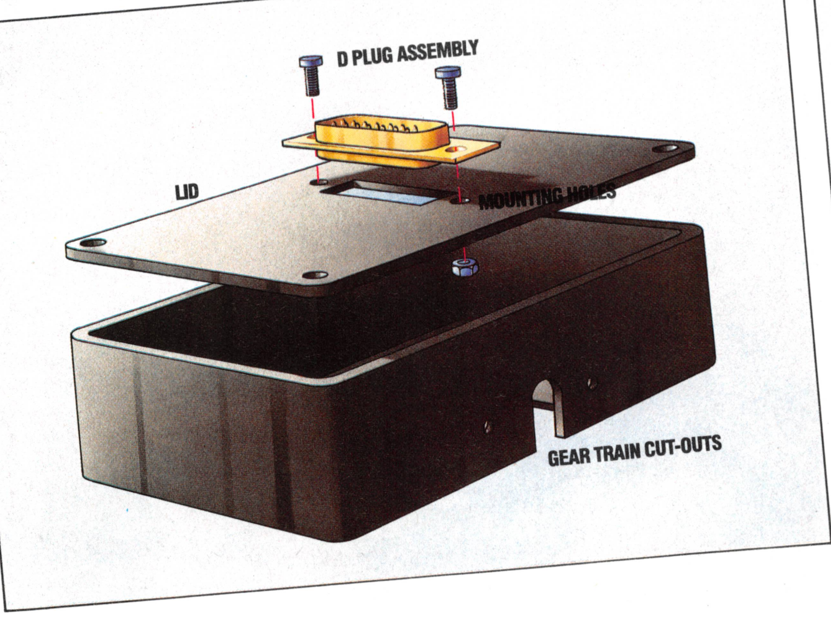
2 And 3



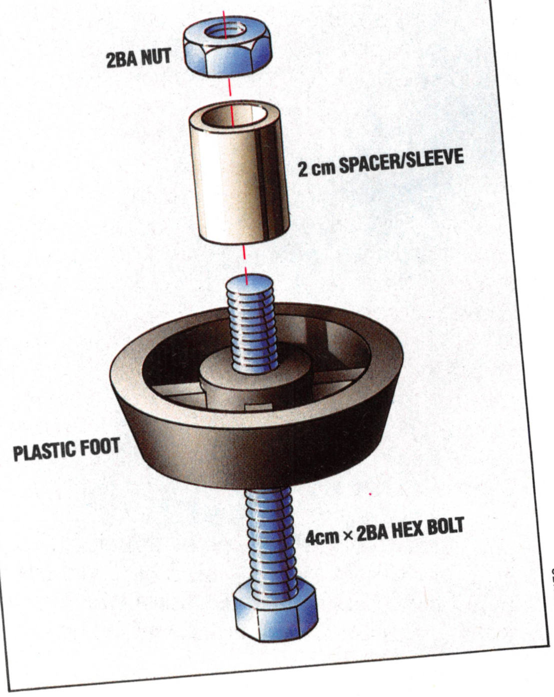
Step Four

Bolt the male (plug) D connector in place in the lid with the pins facing upwards using the 6BA bolts and nuts. Finally mount the two balancing feet, using a 2BA bolt through each foot and fastening it inside the case with a nut. The feet should be spaced with the base of the feet 3cm from the bottom of the case. Spacing can be achieved by placing 2cm sleeves between the foot and the bottom of the case. Alternatively, two lock nuts may be used

4



4



ON LOCATION

So far in our adventure game programming project, we have developed a map of the locations that form the basis of a game and written a utility routine that formats output to the screen. We are now in a position to design routines that describe locations within the game and allow the player to move between locations.

The basic description of each location is held in the array LN$() (see page 767) and can be accessed simply by specifying the number of the location arrived at. In Haunted Forest, the position held by the player at any given time is stored in the variable P, and, therefore, the description of that location is stored in LN$(P). When the location data was first designed the description's final grammatical context was kept in mind; the description always being phrased in such a way that it could be prefixed by 'You are...'. For a given location, P, the description can be formatted and output to the utility developed in the last instalment, by combining 'You are' with the description held for that location in the array LN$(). Line 2010 in the Haunted Forest listing shows this.

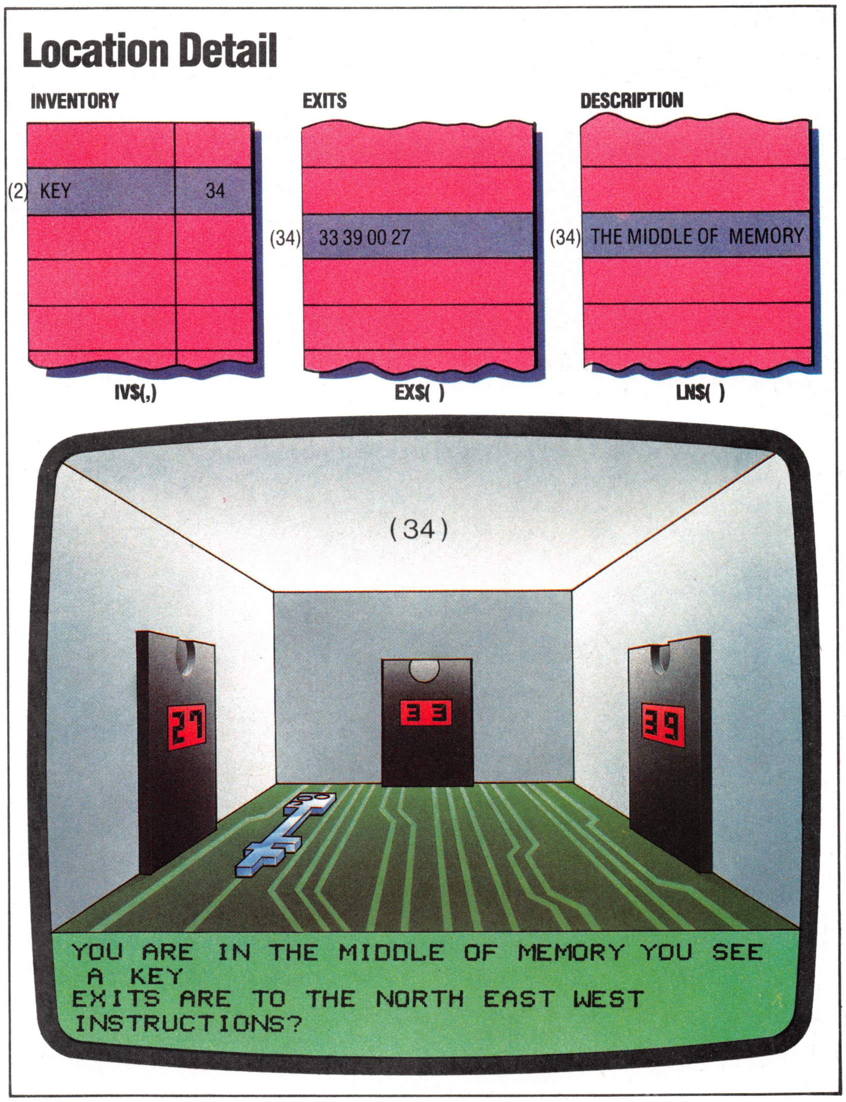
In addition to the basic description of the location arrived at, the player will also want to know if any objects are present. The objects used in the game are stored — together with their initial positions in the inventory — in a two-dimensional array, IV$(,). For example, IV$(N,1) holds the description of the Nth object in the inventory, and IV$(N,2) holds its position. If we wish to determine whether or not there is an object at a particular location we must search through the inventory, checking each object's position against the number of the location that is being described. As there are only three objects in Haunted Forest and eight objects in Digitaya, a simple linear search using a FOR...NEXT loop can be implemented.

Lines 2040-2080 show the search loop used in Haunted Forest. The second column of the inventory array is scanned for a match with the current location, P. When a match is found, then the corresponding description is added to the sentence that describes the objects. As more than one object may be present in any one location, we must allow for the construction of a sentence where a list of objects is given, each separated by a comma. By using SP$, initially as a null string, and later as a comma, we can insert the correct punctuation between each item. A flag, F, initially set to zero, is set to one to signal the fact that an object match has been found during the search. If the flag remains at zero at the end of the search, then no objects are present, and this fact can be output to the player — as in line 2090 of Haunted Forest.

```
2000 REM **** DESCRIBE LOCATION ****
2010 SN$="YOU ARE "+LN$(P):GOSUB5500
2020 SN$="YOU SEE "
2030 REM ** CHECK INVENTORY FOR OBJ **
2040 F=0:SP$=""
2050 FOR I=1 TO 3
2060 IF VAL(IV$(I,2))<>P THEN 2080
2070 SN$=SN$+SP$+"A "+IV$(I,1):F=1:SP$=", "
2080 NEXT I
2090 IF F=0 THEN SN$=SN$+"NO OBJECTS"
2100 GOSUB5500:REM FORMAT OUTPUT
2110 RETURN
```

The data containing details of the possible exits from each location is held in the array EX$(). Each string value is made up of eight digits. By subdividing these eight digits into groups of two, we obtain — working from left to right — the

A Room With A View
The details of the locations in our adventure game are held in three string arrays, which contain object names and whereabouts (V$), location exits (EX$) and descriptions (LN$). EX$ (34), for example, might contain the eight-digit number 33390027, showing that location 34 connects to locations 33,39 and 27 by its north, east and west exits respectively. LN$(34) contains ' The Middle Of Memory', which describes location 34. IV$(2,2) contains the number 34, showing that IV$(2,1) — The Key — is in location 34. Given the current location number the program assembles this information into a description



```
YOU ARE IN THE MIDDLE OF MEMORY YOU SEE
 A KEY
EXITS ARE TO THE NORTH EAST WEST
INSTRUCTIONS?
```

numbers of the locations lying to the north, east, south and west of the current location. In order to determine which exits are possible, the program first splits the eight-digit string into the four numbers that describe which location lies in each direction.

```
2300 REM **** DESCRIBE EXITS S/R ****
2310 EX$=EX$(P)
2320 NR=VAL(LEFT$(EX$,2))
2330 EA=VAL(MID$(EX$,3,2))
2340 SO=VAL(MID$(EX$,5,2))
2350 WE=VAL(RIGHT$(EX$,2))
```

If there is no exit in a given direction, the value assigned is zero — and this is a great help with the description of the exits. A preliminary check must be made to see if any exits are possible before starting to construct the sentence 'There are exits to the . . .'. This can be done by performing a logical OR on all four direction variables, and this will only produce a zero result if all four direction variables are zero. If this is not the case, then the routine continues to test each direction variable in turn. If the variable is non-zero then the corresponding direction is added to the sentence.

```
2355 IF(NR OR EA OR SO OR WE)=0 THEN RETURN
2360 PRINT:SN$="EXITS ARE TO THE "
2370 IF NR <>0 THEN SN$=SN$+"NORTH "
2380 IF EA <>0 THEN SN$=SN$+"EAST "
2390 IF SO <>0 THEN SN$=SN$+"SOUTH "
2400 IF WE <>0 THEN SN$=SN$+"WEST "
2410 GOSUB 5500:REM FORMAT
2415 PRINT
2420 RETURN
```

Now that we have developed routines that describe each location, we can develop procedures that will allow the player to do things within the world we have created. In a future instalment of the project, we shall be considering more detailed algorithms that analyse instructions. For now, we will deal with the movement instructions the player can issue by simply entering a one word direction command, such as 'NORTH' or 'SOUTH'. If such an instruction is passed to a movement subroutine as the variable NN$, then the movement routine is as follows:

```
3500 REM **** MOVE S/R ****
3510 MF=1:REM SET MOVE FLAG
3520 DR$=LEFT$(NN$,1)
3530 IF DR$<>"N"ANDDR$<>"E"ANDDR$<>"S"ANDDR$<>"W"
  THEN GOTO3590
3540 IF DR$="N"AND NR<>0 THEN P=NR:RETURN
3550 IF DR$="E"AND EA<>0 THEN P=EA:RETURN
3560 IF DR$="S"AND SO<>0 THEN P=SO:RETURN
3570 IF DR$="W"AND WE<>0 THEN P=WE:RETURN
3580 PRINT:PRINT"YOU CAN'T ";IS$
3585 MF=0:RETURN
3590 REM ** NOUN NOT DIRECTION **
3600 PRINT"WHAT IS ";NN$;" ?"
3610 MF=0:RETURN
```

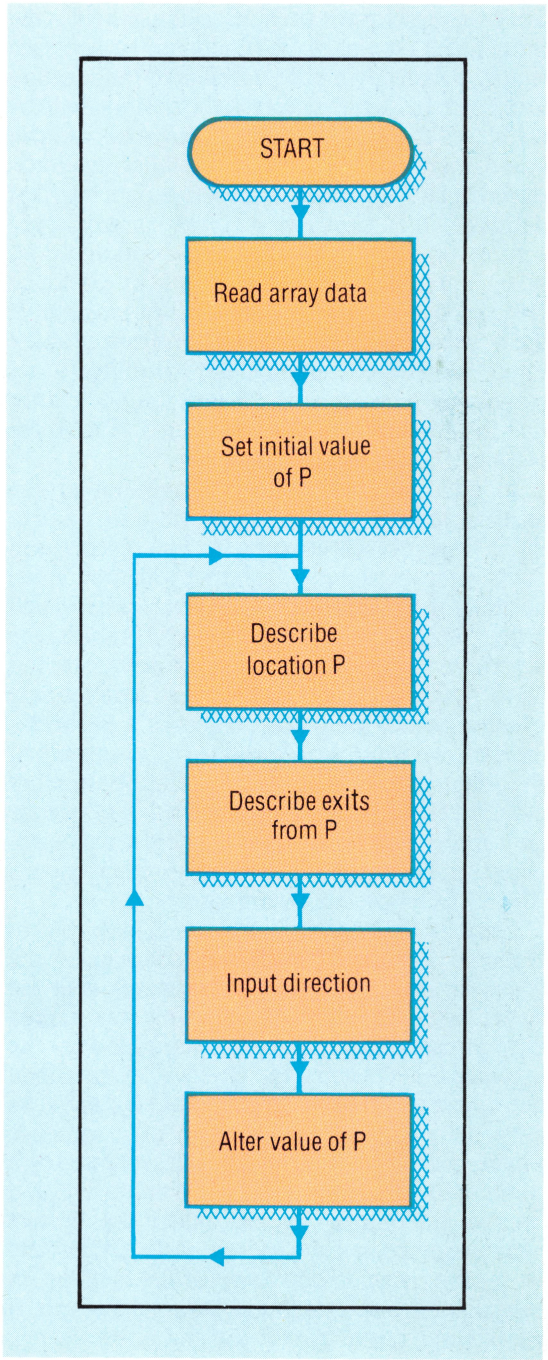
This routine actually uses only the first letter of the direction command passed to it. It begins by checking that the command is, in fact, a direction. If so, the direction specified in the command is acted upon. After ensuring that there is an exit in that direction, P — the variable that keeps track of the player's position — is changed to the value of NR, EA, SO or WE.

Before we can use the subroutines that we have developed here, however, we need to tie them all together to form a repeating loop. The flowchart shows the logical structure of this main calling loop. Although this is not the final structure of the main program loop it serves to demonstrate the aspects of the program covered so far. To use the subroutines given here, insert the following lines, which form a part of the main loop.

```
200 GOSUB6000:REM READ ARRAY DATA
210 P=INT(RND(TI)*10+1):REM START POINT
230 REM **** MAIN LOOP STARTS HERE ****
240 MF=0:REM MOVE FLAG
245 PRINT
250 GOSUB2000:REM DESCRIBE POSITION
255 GOSUB2300:REM DESCRIBE EXITS
260 PRINT:INPUT"INSTRUCTIONS";IS$
```

Also include the following lines in the main calling loop:

```
270 NN$=IS$:GOSUB 3500:REM MOVE
280 GOTO 230:REM RESTART MAIN LOOP
```



SPECTRUM VARIATIONS

Because the Spectrum holds all string arrays as fixed-length strings, problems arise when we wish to print out an element of a string array as part of a larger sentence. When dimensioning an array on the Spectrum, the last number in the statement defines the length of each element in the array. For example, DIM a$(3,2,20) dimensions a three-by-two element array, with each element having a fixed length of 20 characters. If we assign an element in the array to a string with less than 20 characters, then the difference is made up by adding spaces to the end of the string. This wastes precious space in memory. Therefore, to insert Spectrum string-array variables into sentences, we must first of all remove any trailing spaces. Spectrum users should type in the following routine to do this in the Haunted Forest listing:

```
7000 REM **** SPECTRUM TRUNCATE ****
7010 FOR I=LEN(A$) TO 1 STEP -1
7020 IF A$(I TO I)<>" " THEN LET N=I:LET I=1
7030 NEXT I
7040 LET S$=S$+A$(TO N)
7050 RETURN
```

For the Digitaya listing, type in these same commands, but use line numbers 8500 to 8550.

This routine truncates A$, removing any trailing spaces, before adding it to S$. Remember that S$ is the string variable used to assemble a sentence for formatting. To use this routine, we must pass the string-array element (to be incorporated into the sentence) to the variable A$, and then call the subroutine. Therefore, we must make the following alterations to Spectrum versions of Haunted Forest and Digitaya:

Haunted Forest:

```
2010 LET S$="YOU ARE ":A$=L$(P):GOSUB7000:
  GOSUB 5500
2070 LET S$=S$+P$+"A ": A$=V$(I,1):
  GOSUB7000:LET F=1:LET P$=", "
```

Digitaya:

```
1450 LET S$="YOU ARE ":A$=L$(P):
  GOSUB8500:GOSUB5880
1500 IF VAL(V$(I,2))=P THEN LET S$=S$+P$+"A "
  :A$=V$(I,1):GOSUB8500:LET F=1:LET P$=" "
```

Digitaya Listing

The structure of Digitaya is similar to Haunted Forest. Add the following lines to the listings given so far in the project:

```
1100 GOSUB6090:REM READ ARRAY DATA
1210 PRINT:INPUT"INSTRUCTIONS";IS$
1120 P=47:REM START POINT
1130 :
1140 REM **** MAIN LOOP STARTS HERE ****
1150 :
1160 MF=0:PRINT
1170 GOSUB1440:REM DESCRIBE POSITION
1180 GOSUB1560: REM LIST EXITS
```

Also include these lines:

```
1220 NN$=IS$:GOSUB 2000:REM MOVE
1230 GOTO 1140:REM RESTART MAIN LOOP
```

Describe Location And Exits

```
1440 REM **** DESCRIBE POSITION S/R ****
1450 SN$="YOU ARE "+LN$(P):GOSUB5880
1460 SN$="YOU SEE "
1470 REM ** SEARCH FOR OBJECT **
1480 F=0:SP$=""
1490 FOR I=1TO8
1500 IF VAL(IV$(I,2))=P THEN SN$=SN$+SP$+"A
"+IV$(I,1):F=1:SP$="   "
1510 NEXTI
1520 IF F=0 THENSN$=SN$+"NO OBJECTS"
1530 GOSUB5880:REM FORMAT
1540 RETURN
1550 :
1560 REM **** LIST EXITS S/R ****
1570 EX$=EX$(P)
1580 NR=VAL(LEFT$(EX$,2))
1590 EA=VAL(MID$(EX$,3,2))
1600 SO=VAL(MID$(EX$,5,2))
1610 WE=VAL(RIGHT$(EX$,2))
1620 IF(NR OR EA OR SO OR WE)=0THEN RETURN
1630 PRINT:SN$="EXITS ARE TO THE "
1640 IF NR<>0 THEN SN$=SN$+"NORTH "
1650 IF EA<>0 THEN SN$=SN$+"EAST "
1660 IF SO<>0 THEN SN$=SN$+"SOUTH "
1670 IF WE<>0 THEN SN$=SN$+"WEST "
1675 GOSUB 5830:REM FORMAT
1680 PRINT:RETURN
```

Move To Subroutine

```
2000 REM **** MOVE S/R ****
2010 MF=1:REM MOVE FLAG SET
2020 DR$= LEFT$(NN$,1)
2030 IFDR$<>"N"ANDDR$<>"E"ANDDR$<>"S"ANDDR$<>
"W"TH EN2100
2040 IF DR$="N" AND NR<>0 THEN P=NR:RETURN
2050 IF DR$="S" AND SO<>0 THEN P=SO:RETURN
2060 IF DR$="E" AND EA<>0 THEN P=EA:RETURN
2070 IF DR$="W" AND WE<>0 THEN P=WE:RETURN
2080 PRINT"YOU CANT ";IS$
2090 MF=0:RETURN
2100 REM NOUN NOT OK
2110 PRINT"WHAT IS ";NN$;" ?"
2120 MF=0:RETURN
```

Basic Flavours

Spectrum:

Throughout both games listings, replace EX$() with E$(), EX$ with X$, SN$ with S$, IS$ with T$, LN$() with L$(), NN$ with R$, SP$ with P$, DR$ with D$. For the Digitaya listing, substitute the following lines:

```
1580 LET NR=VAL(X$(TO 2))
1590 LET EA=VAL(X$(3 TO 4))
1600 LET SO=VAL(X$(5 TO 6))
1610 LET WE=VAL(X$(7 TO))
2020 LET D$=R$(TO 1)
```

For the Haunted Forest listing, substitute the following lines:

```
210 RANDOMISE:P=INT(RND(1)*10+1)
2320 LET NR=VAL(X$(TO 2))
2330 LET EA=VAL(X$(3 TO 4))
2340 LET SO=VAL(X$(5 TO 6))
2350 LET WE=VAL(X$(7 TO))
3520 LET D$=R$(TO 1)
```

BBC Micro:

Substitute the following line in the Haunted Forest listing:

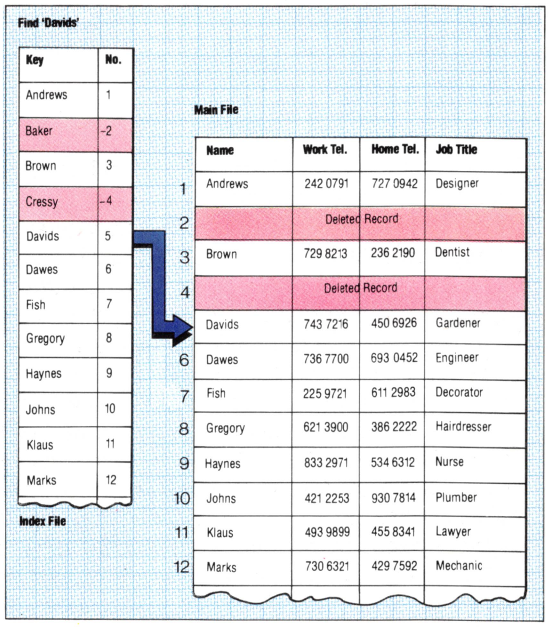
```
210 P=RND(10)
```

I

INDEX
The word *index* has several different definitions in computing. The first refers to a number or item of data in a list that indicates where a piece of information can be located. For example, a BASIC array contains a number of different elements, and each of these can be separately accessed by selecting a number associated with that element. This number is referred to as 'the index'. In machine code, an index is a number held in an *index register*. This index is the number that must be added to a particular address, which then points to another address — the contents of which are to be modified. This method of addressing is used for processing arrays.

INDEXED FILE
As its name implies, an *indexed file* is simply a file whose organisation is dependent on an index. Indexed files may be organised so that the index is separated from the file itself — as in a book index. In computing, this type of indexed file may be found on a floppy disk, on which a specific track is set aside for the directory of the disk's contents. Each time the disk is accessed the directory is examined to discover the required file's location.

A common and important file type is the *indexed sequential file*. Here, a sequential file is stored in sorted order, and an index file is created from it, consisting of the sort key field from each record of the original file, in the sorted order. Records are located by searching the index file for the desired key field; its position in the index is the same as the parent record's position in the sequential file. This can now be accessed reasonably quickly by skipping the appropriate number of records from the start of the file. The technique was first developed for mainframe tape-based systems in which the whole index file could usually be held and searched quickly in memory; the parent file would generally contain too many lengthy records to fit into available memory.



Key	No.
Andrews	1
Baker	-2
Brown	3
Cressy	-4
Davids	5
Dawes	6
Fish	7
Gregory	8
Haynes	9
Johns	10
Klaus	11
Marks	12

	Name	Work Tel.	Home Tel.	Job Title
1	Andrews	242 0791	727 0942	Designer
2	Deleted Record			
3	Brown	729 8213	236 2190	Dentist
4	Deleted Record			
	Davids	743 7216	450 6926	Gardener
6	Dawes	736 7700	693 0452	Engineer
7	Fish	225 9721	611 2983	Decorator
8	Gregory	621 3900	386 2222	Hairdresser
9	Haynes	833 2971	534 6312	Nurse
10	Johns	421 2253	930 7814	Plumber
11	Klaus	493 9899	455 8341	Lawyer
12	Marks	730 6321	429 7592	Mechanic

KEVIN JONES

Index Linked
The index file consists of the key fields of the data records, sorted in the same order as the main file. A record is located by searching the index file for the key and then skipping the appropriate number of records in the main file. Records are deleted temporarily by marking them in the index file; from time to time the main file willl be re-sorted to take account of changes, and the deleted records will then be permanently removed

INDEX REGISTER
An *index register* is an area set aside within the central processing unit (CPU) of a microcomputer for the storage of the index currently being used by the program. This is particularly useful for 'indexed addressing'. In this method of addressing, the number held in the index register is added to the address specified in the instruction, and the sum of the two numbers is the address to be accessed by the computer. For example, the instruction LDA BASE,X will LoaD the Accumulator with the contents of the address whose value is equal to BASE + the contents of register X.

This technique is mostly used to access a table of numbers (an array). A number of instructions, or op-codes, are set aside specifically to manipulate the numbers held within index registers. An index register may also be used as a general-purpose register. This means that the register can be used by the programmer as a short-term storage device for numbers. Used in this way, an index register can cut both memory use and processing time appreciably. The alternative is to have the processor store the number in RAM — a process that is time-consuming and occupies valuable memory space.

INFORMATION HIDING
Information hiding is a concept relating to structured programming. The principle behind this method of programming is that a program should be constructed of individual modules that are self-contained, and which can easily be understood and modified. Furthermore, the information and decisions taken within a module should, as far as possible, be exclusive to that module. This is known as 'information hiding' because the information or decision is 'hidden' from the rest of the program. This concept was first developed by David Parnas, who proposed that all modules within a program should ideally contain only one decision.

INFORMATION MANAGEMENT SYSTEM
An *information management system* is designed to deal with the organisation of information within a system. It is often used with databases, which are themselves ways of organising data. The information must be stored in such a way as to allow fast and easy access by the user. This means that the system must not only be able to store the information but must also maintain an index system to allow that information to be retrieved efficiently. However, an information management system is more than simply the location, storage, retrieval and cataloguing of data. Although databases are within the field of IMS, the term is more broadly based, encompassing such features as the computer's operating system and memory management, as well as other programs in which data is constantly being changed and updated. IMS, combined with distributed processing, is the most important influence on data processing.

LAST ORDERS

There are three commands for our debugging program that are yet to be designed. Before we look at these, however, we will consider the interrupt mechanism used to transfer control between the debugging program and the program being debugged at the breakpoints. We will also design the initialisation procedure.

The interrupt mechanism is used at breakpoints in the original program, where we have replaced an instruction with an SWI (SoftWare Interrupt) op-code. The SWI, like the other interrupts on the 6809, is vectored through a specific memory location — namely, $FFFA. This means that when an SWI is executed the registers are saved on the stack and the processor loads the 16-bit address at $FFFA and $FFFB into the program counter (PC). Execution then continues from that address. Our task is to change this vector so that it points to the entry point of our debugger program. One problem here is that interrupt vectors are almost always held in ROM. The fact that these addresses are fixed, therefore, means that the operating system must have some other means of vectoring interrupts.

The normal system is to have a jump table (see page 639) held in an area of 'scratchpad' RAM, which is memory that is not normally available to programs but is reserved for use by the operating system. The address pointed at by the vector contains a JMP instruction followed by an address, which normally will point back into the operating system. However, we can change this address to the one we want so the first instruction executed after the software interrupt will be a JMP to the entry address of the debugger. We must be careful to replace the original contents of the jump table before our program finishes executing, because it is always possible that the operating system will execute an SWI subsequently. It is worth remembering that the 6809 has three software interrupts, and there is no reason why either SWI2 (op-code 10 3F and vector at $FFF4) or SWI3 (op-code 11 3F and vector at $FFF2) should not be used — although the fact that these use two-byte op-codes makes some changes necessary in the debugger program.

A further problem is that our program can only occupy whatever memory is left free by the program we are debugging. The debugger must therefore be relocatable. You will have noticed that all references to memory locations in the program have been (or should have been) made using program counter relative addressing. The problem is that at some point we must know the absolute address of the program entry point so that we can place it in the interrupt jump table. This address must be calculated at run-time, since the assembler cannot deal with it.

Our first task then is calculating this address and inserting it into the jump table. Note that the entry point address for SWI will be different from the start address of the debugger program, because the routine at the program start address must handle this initialisation procedure, which will not be needed when we re-enter the program via SWI. Accordingly, we will handle all the initialisation within a subroutine; the entry point will then be the address containing the instruction after the BSR call to this subroutine. Very conveniently, this address is precisely the one saved on the stack by the BSR call so we can read it from the stack in order to place it at the appropriate point in the jump table.

The other job of this initialisation procedure is to obtain the start address of the program to be debugged. Here is the completed design:

INITIALISATION PROCEDURE

Data:

```
    Vector-Address is the address to be found at
    $FFFA in X
    JMP-Opcode is the op-code for the JMP
    instruction in A
    Entry-Address is the address of the entry point in Y
    Start-Address of the program to be debugged in D
Process:
    Get Vector-Address
    Store JMP-Opcode at Vector-Address
    Get Entry-Address
    Store it at (Vector-Address + 1)
    Get Start-Address from keyboard
    Save it
```

We can now return and complete the coding of the three remaining commands. A further point to consider involves one of these commands — namely command R, which displays the contents of the registers. We do not, of course, want to display the current contents of the registers while the debugger is running; instead, we want to look at the contents of the registers as they were when the breakpoint occurred. This means that we want to look at the values that were placed on the stack by the SWI instruction. However, there will be other values placed on top of these on the stack by the time we want to get at them. We could probably calculate the number of unwanted bytes on the stack and obtain the register values by discarding this amount. But a simpler solution is to

B	insert Breakpoint
U	Un-insert (remove breakpoint)
D	Display current breakpoints
S	Start running program
G	Go (resume from where the program left off)
R	display contents of Registers
M	inspect and change Memory location
Q	Quit

save the value of the stack pointer as the first operation after the interrupt occurs, so that it can be used as a reference.

In coding the R command, we will assume that this has been done, so that we can retrieve these register contents. The structure of the routine is perfectly straightforward — we simply take each value in turn without actually pulling them off the stack and display them with appropriate labels. The only exception will be the value of S — this should be the value prior to the interrupt and can be obtained by adding the appropriate amount to the saved value of S that we use to reference the stacked register values.

COMMAND R

Data:

Stack-Pointer is the value of the top of stack after interrupt in X
Single-Byte-Value holds the values of single-byte registers in B
Two-Byte-Value holds the values of 16-bit registers in D
Labels holds the labels for the nine registers

```
Get Stack-Pointer
Load CC into Single-Byte-Value
Display label(1), Single-Byte-Value
Repeat the above for A, B, and DP
Load X into Two-Byte-Value
Display label(5), Two-Byte-Value
Repeat the above for Y, U and PC
Add 12 to original value of Stack-Pointer
Display label(9), Stack-Pointer
```

There are two remaining commands: Q, to quit the program, does not need a special routine of its own; and G, to resume program execution after a breakpoint. At this point we have to replace the SWI instruction that caused the break with the original instruction that it replaced and then pass control back to that instruction. We can restore the registers to their original contents easily enough, simply by using an RTI, which unstacks them all. We must, however, be careful that the value of the PC that is unstacked is going to be the value for the next instruction; since this is one greater than the value we require, we must adjust the value on the stack before we return.

COMMAND G

Data:

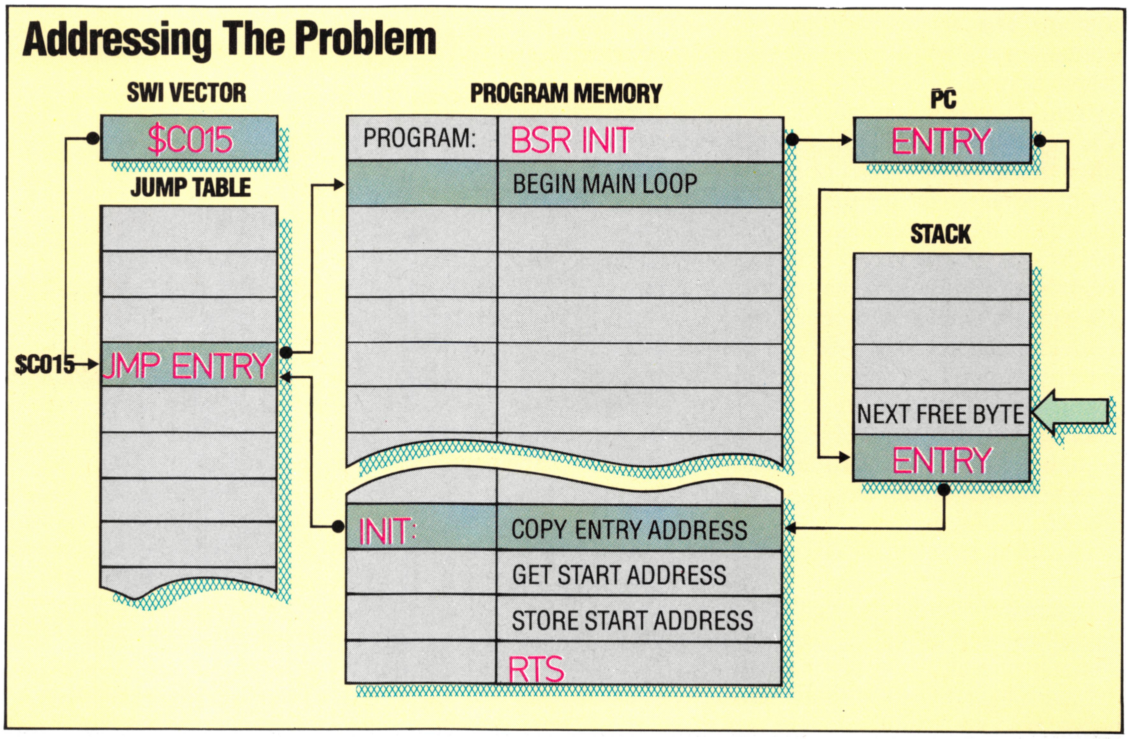
Breakpoint-Table is a table of 16-bit addresses of breakpoints
Removed-Values is a table of op-codes replaced with SWIs
Next-Breakpoint is a number in the range 1 to 16
Stack-Pointer is the saved value of the stack pointer after the SWI

Process:

```
If Next-Breakpoint >0 and <=16 then
    Get op-code from Removed-Values (Next-Breakpoint)
    Store it at address in Breakpoint-Table (Next-Breakpoint)
    Set S to Stack-Pointer
    Decrement value of PC on stack
    Increment Next-Breakpoint
    Return from interrupt
else
    Return from subroutine
```

Our 6809 machine code series concludes in the next instalment, when we code the main module of our debugger, and look at the operation of the program as a whole.

A Stack In Time
The debugger program begins with a BSR call to the initialisation routine, followed by the start of the main program loop. One of the initialisation tasks is to ascertain the absolute address of this loop start, and to copy it into the interrupt jump table so that when an SWI is executed control will pass through the jump table and back to the loop start. This address cannot be known in advance because the program must be fully relocatable; fortunately, the return address stacked by the BSR is precisely the address in question, so the initialisation routine needs merely to copy it from the stack to the jump table



Initialisation Procedure

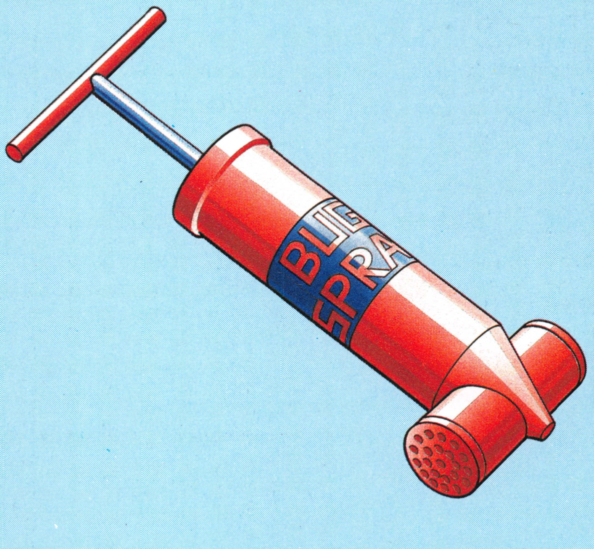
```
START    RMB    2               To save the start address
OPJMP    FCB    $0E             JMP-opcode
INIT     LDX    $FFFA           Get Vector-Address
         LDA    OPJMP, PCR      Get JMP-opcode and
         STA    , X+            save it at Vector-Address
         LDY    1,S             Get Entry-Address from the stack
         STY    ,X              Save it at Vector-Address + 1
         BSR    GETADD          Get Start-Address from keyboard
         STD    START,PCR       Save it
         RTS                    Return
```

Command R

```
STACKP   RMB    2               Stack-Pointer
LABELS   FCC    'CC A BDP X Y
                     UPC S'
SPACE    FCB    32              ASCII code for space
CMDR     PSHS   A,B,X,Y         Save used registers
         LDX    STACKP, PCR     Get Stack-Pointer
         LEAY   LABELS, PCR     Use Y to point to label
         LDA    #4              Number of single byte registers
FOR01    BSR    CMDR1           Display next register
         DECA                   four times
         BGT    FOR01
         LDA    #4              Number of two byte registers
FOR02    BSR    CMDR2           Display next register
         DECA                   four times
         BGT    FOR02
         LDA    ,Y+             First character of label
         BSR    OUTCH           Display it
         LDA    ,Y+             Second character of label
         BSR    OUTCH           Display it
         LDA    SPACE,PCR       Display space
         BSR    OUTCH
         TFR    X,D             X now contains the required
                                value of S
         BSR    DSPADD          Display S
         PULS   A,B,X,Y,PC      Restore and return
       *Subroutine to display a single byte register
CMDR1    PSHS   A               Save A
         LDA    ,Y+             First character of label
         BSR    OUTCH           Display it
         LDA    ,Y+             Second character of label
         BSR    OUTCH           Display it
         LDA    SPACE,PCR       Display space
         BSR    OUTCH
         LDB    ,X+             Get next register into Single-Byte
                                -Value
         BSR    DSPVAL          Display Single-Byte-Value
         PULS   A,PC            Restore A and return
       *Subroutine to display a two byte register
CMDR2    PSHS   A               Save A
         LDA    ,Y+             First character of label
         BSR    OUTCH           Display it
         LDA    ,Y+             Second character of label
         BSR    OUTCH           Display it
         LDA    SPACE,P-CR      Display space
         BSR    OUTCH
         LDD    ,X++            Get next register into Two-Byte
                                -Value
         BSR    DSPADD          Display Two-Byte-Value
         PULS   A,PC            Restore A and return
```

Command G

```
BPTAB    RMB    32              Breakpoint-Table
REMTAB   RMB    16              Removed-Values
NEXTBP   RMB    1               Next-Breakpoint
CMDG     PSHS   A               Save A in case we do a normal return
         LDA    NEXTBP,PCR      Next-Breakpoint
IF04     BLE    ENDF04          If Next-Breakpoint >0
         CMPA   MAXBP,PCR       and <=16
         BGT    ENDF04          (maximum number of
                                breakpoints)
         DECA                   Convert to offset into table
         LEAX   BPTAB,PCR       Address of Breakpoint- Table
         LEAY   REMTAB,PCR      Address of Removed- Values
         LDB    A,Y             Get Removed-Value
         LSLA                   Convert A to offset for 16-bit table
         STB    [A,X]           Store it at address in Breakpoint-
                                Table
         LDS    STACKP,PCR      Get Stack-Pointer into S
         DEC    10,S            Adjust value of PC on stack
         INC    NEXTBP,PCR      Increment Next-Breakpoint
         RTI                    Return from interrupt
ENDF04   PULS   A,PC            Restore and return
```



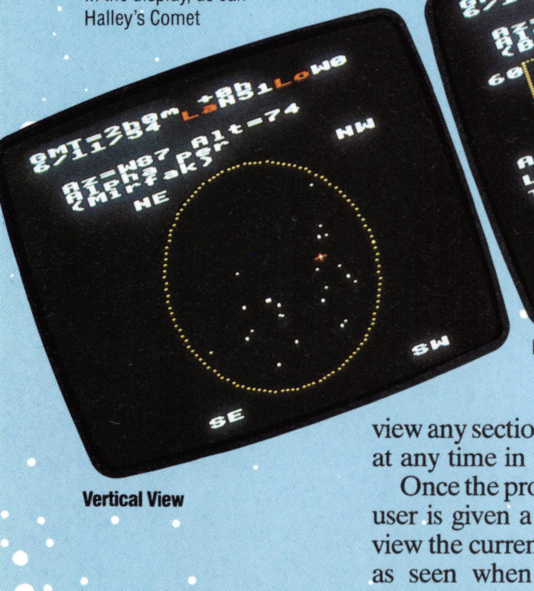
STARS ON SCREEN

The BBC Micro is the machine used in the majority of British schools, and it is therefore hardly surprising that a large amount of 'educational' software has been developed for it. Here we look at one such package — Starfinder, a program that is designed for amateur astronomers.

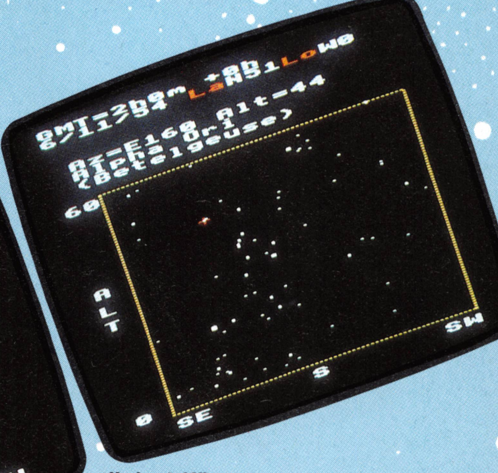
Astronomers, especially the British variety, have always faced one problem in particular — that of the weather. It is not uncommon for professional astronomers to spend weeks preparing for a particularly stunning event, only to discover that dense cloud has obscured the view. This is the reason that most observatories are now built at high altitude, in places where there is little cloud, or even in space.

Century Software has now brought the universe to the small screen with Starfinder, a package containing cassette software and a book. In essence, the Starfinder program allows the user to view any section of the sky in any part of the world at any time in the 20th century.

Starlight, Starbright
Starfinder's two display modes give a rectangular or circular skyscape depending upon whether the line of sight is horizontal or vertical. The star scene can be adjusted for a point of view anywhere on the Earth's surface at any time in the 20th century. Particular stars or planets can be searched for in the display, as can Halley's Comet



Vertical View



Horizontal View

Once the program has loaded from cassette, the user is given a list of options. One of these is to view the current sky; the default view is of the sky as seen when looking south from London at midnight (Greenwich Mean Time) on 21 November 1984. This seems to be purely arbitrary, as there is nothing special about the view of the sky on this date: it's likely that this was designed to coincide with the package's launch date rather than with a specific cosmic event.

At the top of the screen, the program displays the time and date, and gives the observer's location in longitude and latitude. Below this information is the star chart itself, which shows the night sky from the south-west to the south-east, assuming an altitude (the angle of view) of 60°. The program takes a few seconds to display the stars themselves, as a large amount of number-crunching is required to process and then plot each of the stars.

The stars are shown as white squares (the program runs in mode 4), and brighter stars are depicted as being larger. It is a pity that the Acorn machines do not have a 'brightness' command, which would make the display more realistic. Planets are also square-shaped, but are plotted in red, while the sun is represented by a large yellow square and the moon by a small yellow point. By using the 'space probe', which is moved by using the cursor keys, it is possible to identify any of the stars shown. When the space probe (a red cross) is positioned over a star, that star's technical designation and popular name are shown above the map, together with its co-ordinates, given as an altitude and an azimuth figure (the altitude is expressed as a positive or negative figure, with the horizon as zero; while the azimuth shows the number of degrees east or west of due north). The view may be changed from the keyboard by altering either of these figures.

By returning to the main menu, the user may change the view to correspond to that revealed at any time, anywhere in the world. It is also possible to use the program to find a particular heavenly body: a star, a planet or Halley's Comet.

Although this package is comprehensive, it does have limitations. The primary restriction is on the number of stars that may be displayed. The programmer, Robert Alpiar, has chosen to include stars of a magnitude of 4 or lower ('magnitude' refers to a star's brightness, with a high magnitude representing a faint star). The naked eye is capable of discerning stars of magnitude 6, so the program does not claim to show as many stars as a person might see on a clear night.

The book that is included with the program explains the use of Starfinder, but also includes other tips for amateur astronomers, including advice on different types of telescope and the best times to view planets and stars.

Starfinder: For the BBC Micro and the Electron £12.95.
Publishers: Century Software, Portland House, 12-13 Greek Street, London, W1V 5LE.
Authors: Book by Heather Couper, program by Ronald Alpiar.
Joysticks: Not Required.
Format: Cassette.

DATABASE

Here, courtesy of Zilog Inc., we publish the final part of the Z80 programmers' reference card.

Call and Return Groups and Restart

Call and Return Group

			CONDITION									
			UN COND.	CARRY	NON CARRY	ZERO	NON ZERO	PARITY EVEN	PARITY ODD	SIGN NEG.	SIGN POS.	REG. B≠0
'CALL'	IMMEDIATE EXTENSION	nn	CD n n	DC n n	D4 n n	CC n n	C4 n n	EC n n	E4 n n	FC n n	F4 n n	
RETURN 'RET'	REGISTER INDIRECT	(SP) (SP + 1)	C9	D8	D0	C8	C0	E8	E0	F8	F0	
RETURN FROM INT 'RETI'	REGISTER INDIRECT	(SP) (SP + 1)	ED 4D									
RETURN FROM NON MASKABLE INT 'RETN'	REGISTER INDIRECT	(SP) (SP + 1)	ED 45									

Note: Certain flags have more than one purpose.
Refer to the Z80 CPU Technical Manual for details.

Restart Group

		OP CODE	
CALL ADDRESS	0000_H	C7	'RST 0'
	0008_H	CF	'RST 8'
	0010_H	D7	'RST 16'
	0018_H	DF	'RST 24'
	0020_H	E7	'RST 32'
	0028_H	EF	'RST 40'
	0030_H	F7	'RST 48'
	0038_H	FF	'RST 56'

Mnemonic	Symbolic Operation	S	Z		H		P/V	N	C	Opcode 76 543 210	Hex	No.of Bytes	No.of M Cycles	No.of T States	Comments
CALL nn	$(SP-1) \leftarrow PC_H$ $(SP-2) \leftarrow PC_L$ $PC \leftarrow nn$	•	•	X	•	X	•	•	•	11 001 101 ← n → ← n →	CD	3	5	17	
CALL cc, nn	If condition cc is false continue, otherwise same as CALL nn	•	•	X	•	X	•	•	•	11 cc 100 ← n → ← n →		3 3	3 5	10 17	If cc is false If cc is true
RET	$PC_L \leftarrow (SP)$ $PC_H \leftarrow (SP+1)$	•	•	X	•	X	•	•	•	11 001 001	C9	1	3	10	
RET cc	If condition cc is false continue, otherwise same as RET	•	•	X	•	X	•	•	•	11 cc 000		1 1	1 3	5 11	If cc is false. If cc is true. cc Condition 000 NZ non-zero 001 Z zero 010 NC non-carry 011 C carry 100 PO parity odd 101 PE parity even 110 P sign positive 111 M sign negative
RETI	Return from interrupt	•	•	X	•	X	•	•	•	11 101 101 01 001 101	ED 4D	2	4	14	
RETN[1]	Return from non-maskable interrupt	•	•	X	•	X	•	•	•	11 101 101 01 000 101	ED 45	2	4	14	
RST p	$(SP-1) \leftarrow PC_H$ $(SP-2) \leftarrow PC_L$ $PC_H \leftarrow 0$ $PC_L \leftarrow p$	•	•	X	•	X	•	•	•	11 t 111		1	3	11	t p 000 00H 001 08H 010 10H 011 18H 100 20H 101 28H 110 30H 111 38H

NOTE [1]RETN loads $IFF_2 \rightarrow IFF_1$

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ = flag is affected according to the result of the operation.